Chapter 1

Introduction

1.1 Scope

The scope of IEEE Std 1003.1-200x is described in the Base Definitions volume of IEEE Std 1003.1-200x.

1.2 Conformance

Conformance requirements for IEEE Std 1003.1-200x are defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance.

1.3 Normative References

Normative references for IEEE Std 1003.1-200x are defined in the Base Definitions volume of IEEE Std 1003.1-200x.

1.4 Change History

Change history is described in the Rationale (Informative) volume of IEEE Std 1003.1-200x, and in the CHANGE HISTORY section of reference pages.

1.5 Terminology

This section appears in the Base Definitions volume of IEEE Std 1003.1-200x, but is repeated here for convenience:

For the purposes of IEEE Std 1003.1-200x, the following terminology definitions apply:

can

Describes a permissible optional feature or behavior available to the user or application. The feature or behavior is mandatory for an implementation that conforms to IEEE Std 1003.1-200x. An application can rely on the existence of the feature or behavior.

implementation-defined

Describes a value or behavior that is not defined by IEEE Std 1003.1-200x but is selected by an implementor. The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence of the value or behavior. An application that relies on such a value or behavior cannot be assured to be portable across conforming implementations.

The implementor shall document such a value or behavior so that it can be used correctly by an application.

legacy

Describes a feature or behavior that is being retained for compatibility with older applications, but which has limitations which make it inappropriate for developing portable

applications. New applications should use alternative means of obtaining equivalent functionality.

may

Describes a feature or behavior that is optional for an implementation that conforms to IEEE Std 1003.1-200x. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

To avoid ambiguity, the opposite of *may* is expressed as *need not*, instead of *may not*.

shall

For an implementation that conforms to IEEE Std 1003.1-200x, describes a feature or behavior that is mandatory. An application can rely on the existence of the feature or behavior.

For an application or user, describes a behavior that is mandatory.

should

For an implementation that conforms to IEEE Std 1003.1-200x, describes a feature or behavior that is recommended but not mandatory. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

For an application, describes a feature or behavior that is recommended programming practice for optimum portability.

undefined

Describes the nature of a value or behavior not defined by IEEE Std 1003.1-200x which results from use of an invalid program construct or invalid data input.

The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

unspecified

Describes the nature of a value or behavior not specified by IEEE Std 1003.1-200x which results from use of a valid program construct or valid data input.

The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

1.6 Definitions

Concepts and definitions are defined in the Base Definitions volume of IEEE Std 1003.1-200x.

1.7 Relationship to Other Formal Standards

Great care has been taken to ensure that this volume of IEEE Std 1003.1-200x is fully aligned with the following standards:

ISO C (1999)
ISO/IEC 9899: 1999, Programming Languages — C.

Parts of the ISO/IEC 9899: 1999 standard (hereinafter referred to as the ISO C standard) are referenced to describe requirements also mandated by this volume of IEEE Std 1003.1-200x. Some functions and headers included within this volume of IEEE Std 1003.1-200x have a version in the ISO C standard; in this case CX markings are added as appropriate to show where the ISO C standard has been extended (see Section 1.8.1). Any conflict between this volume of IEEE Std 1003.1-200x and the ISO C standard is unintentional.

This volume of IEEE Std 1003.1-200x also allows, but does not require, mathematics functions to support IEEE Std 754-1985 and IEEE Std 854-1987.

1.8 Portability

Some of the utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x and functions in the System Interfaces volume of IEEE Std 1003.1-200x describe functionality that might not be fully portable to systems meeting the requirements for POSIX conformance (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance).

Where optional, enhanced, or reduced functionality is specified, the text is shaded and a code in the margin identifies the nature of the option, extension, or warning (see Section 1.8.1). For maximum portability, an application should avoid such functionality.

1.8.1 Codes

Margin codes and their meanings are listed in the Base Definitions volume of IEEE Std 1003.1-200x, but are repeated here for convenience:

ADV Advisory Information
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the ADV margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the ADV margin legend.

AIO Asynchronous Input and Output
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the AIO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the AIO margin legend.

BAR Barriers
The functionality described is optional. The functionality described is also an extension to the

ISO C standard.

Where applicable, functions are marked with the BAR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the BAR margin legend.

BE Batch Environment Services and Utilities
The functionality described is optional.

Where applicable, utilities are marked with the BE margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the BE margin legend.

CD C-Language Development Utilities
The functionality described is optional.

Where applicable, utilities are marked with the CD margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the CD margin legend.

CPT Process CPU-Time Clocks
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the CPT margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the CPT margin legend.

CS Clock Selection
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the CS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the CS margin legend.

CX Extension to the ISO C standard
The functionality described is an extension to the ISO C standard. Application writers may make use of an extension as it is supported on all IEEE Std 1003.1-200x-conforming systems.

With each function or header from the ISO C standard, a statement to the effect that ‘‘any conflict is unintentional’’ is included. That is intended to refer to a direct conflict. IEEE Std 1003.1-200x acts in part as a profile of the ISO C standard, and it may choose to further constrain behaviors allowed to vary by the ISO C standard. Such limitations are not considered conflicts.

FD FORTRAN Development Utilities
The functionality described is optional.

Where applicable, utilities are marked with the FD margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the FD margin legend.

FR FORTRAN Runtime Utilities
The functionality described is optional.

Where applicable, utilities are marked with the FR margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the FR margin legend.

FSC File Synchronization
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the FSC margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the FSC margin legend.

IP6 IPV6
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the IP6 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the IP6 margin legend.

MC1 Advisory Information and either Memory Mapped Files or Shared Memory Objects
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

This is a shorthand notation for combinations of multiple option codes.

Where applicable, functions are marked with the MC1 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MC1 margin legend.

Refer to the Base Definitions volume of IEEE Std 1003.1-200x, Section 1.5.2, Margin Code Notation.

MC2 Memory Mapped Files, Shared Memory Objects, or Memory Protection
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

This is a shorthand notation for combinations of multiple option codes.

Where applicable, functions are marked with the MC2 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MC2 margin legend.

Refer to the Base Definitions volume of IEEE Std 1003.1-200x, Section 1.5.2, Margin Code Notation.

MF Memory Mapped Files
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MF margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MF margin legend.

ML Process Memory Locking
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the ML margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the ML margin legend.

MLR Range Memory Locking
The functionality described is optional. The functionality described is also an extension to the

ISO C standard.

Where applicable, functions are marked with the MLR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MLR margin legend.

MON Monotonic Clock

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MON margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MON margin legend.

MPR Memory Protection

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MPR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MPR margin legend.

MSG Message Passing

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MSG margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MSG margin legend.

MX IEC 60559 Floating-Point Option

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MX margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MX margin legend.

OB Obsolescent

The functionality described may be withdrawn in a future version of this volume of IEEE Std 1003.1-200x. Strictly Conforming POSIX Applications and Strictly Conforming XSI Applications shall not use obsolescent features.

OF Output Format Incompletely Specified

The functionality described is an XSI extension. The format of the output produced by the utility is not fully specified. It is therefore not possible to post-process this output in a consistent fashion. Typical problems include unknown length of strings and unspecified field delimiters.

OH Optional Header

In the SYNOPSIS section of some interfaces in the System Interfaces volume of IEEE Std 1003.1-200x an included header is marked as in the following example:

OH
```
#include <sys/types.h>
#include <grp.h>
struct group *getgrnam(const char *name);
```

This indicates that the marked header is not required on XSI-conformant systems.

PIO Prioritized Input and Output
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the PIO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the PIO margin legend.

PS Process Scheduling
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the PS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the PS margin legend.

RS Raw Sockets
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the RS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the RS margin legend.

RTS Realtime Signals Extension
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the RTS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the RTS margin legend.

SD Software Development Utilities
The functionality described is optional.

Where applicable, utilities are marked with the SD margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the SD margin legend.

SEM Semaphores
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SEM margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SEM margin legend.

SHM Shared Memory Objects
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SHM margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SHM margin legend.

SIO Synchronized Input and Output
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SIO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SIO margin legend.

SPI Spin Locks

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SPI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SPI margin legend.

SPN Spawn

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SPN margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SPN margin legend.

SS Process Sporadic Server

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SS margin legend.

TCT Thread CPU-Time Clocks

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TCT margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TCT margin legend.

TEF Trace Event Filter

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TEF margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TEF margin legend.

THR Threads

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the THR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the THR margin legend.

TMO Timeouts

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TMO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TMO margin legend.

TMR Timers
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TMR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TMR margin legend.

TPI Thread Priority Inheritance
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TPI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPI margin legend.

TPP Thread Priority Protection
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TPP margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPP margin legend.

TPS Thread Execution Scheduling
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TPS margin legend for the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPS margin legend.

TRC Trace
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TRC margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRC margin legend.

TRI Trace Inherit
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TRI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRI margin legend.

TRL Trace Log
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TRL margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRL margin legend.

TSA Thread Stack Address Attribute
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSA margin legend for the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSA margin legend.

TSF Thread-Safe Functions
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSF margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSF margin legend.

TSH Thread Process-Shared Synchronization
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSH margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSH margin legend.

TSP Thread Sporadic Server
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSP margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSP margin legend.

TSS Thread Stack Address Size
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSS margin legend.

TYM Typed Memory Objects
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TYM margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TYM margin legend.

UP User Portability Utilities
The functionality described is optional.

Where applicable, utilities are marked with the UP margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the UP margin legend.

XSI Extension
The functionality described is an XSI extension. Functionality marked XSI is also an extension to the ISO C standard. Application writers may confidently make use of an extension on all systems supporting the X/Open System Interfaces Extension.

If an entire SYNOPSIS section is shaded and marked XSI, all the functionality described in that reference page is an extension. See the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.439, XSI.

XSR XSI STREAMS

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the XSR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the XSR margin legend.

1.9 Format of Entries

The entries in Chapter 3 are based on a common format as follows. The only sections relating to conformance are the SYNOPSIS, DESCRIPTION, RETURN VALUE, and ERRORS sections.

NAME

This section gives the name or names of the entry and briefly states its purpose.

SYNOPSIS

This section summarizes the use of the entry being described. If it is necessary to include a header to use this function, the names of such headers are shown, for example:

```
#include <stdio.h>
```

DESCRIPTION

This section describes the functionality of the function or header.

RETURN VALUE

This section indicates the possible return values, if any.

If the implementation can detect errors, ‘‘successful completion’’ means that no error has been detected during execution of the function. If the implementation does detect an error, the error is indicated.

For functions where no errors are defined, ‘‘successful completion’’ means that if the implementation checks for errors, no error has been detected. If the implementation can detect errors, and an error is detected, the indicated return value is returned and *errno* may be set.

ERRORS

This section gives the symbolic names of the error values returned by a function or stored into a variable accessed through the symbol *errno* if an error occurs.

‘‘No errors are defined’’ means that error values returned by a function or stored into a variable accessed through the symbol *errno*, if any, depend on the implementation.

EXAMPLES

This section is non-normative.

This section gives examples of usage, where appropriate. In the event of conflict between an example and a normative part of this volume of IEEE Std 1003.1-200x, the normative material is to be taken as correct.

APPLICATION USAGE

This section is non-normative.

This section gives warnings and advice to application writers about the entry. In the event of conflict between warnings and advice and a normative part of this volume of IEEE Std 1003.1-200x, the normative material is to be taken as correct.

RATIONALE

This section is non-normative.

This section contains historical information concerning the contents of this volume of IEEE Std 1003.1-200x and why features were included or discarded by the standard developers.

FUTURE DIRECTIONS

This section is non-normative.

This section provides comments which should be used as a guide to current thinking; there is not necessarily a commitment to adopt these future directions.

SEE ALSO

This section is non-normative.

This section gives references to related information.

CHANGE HISTORY

This section is non-normative.

This section shows the derivation of the entry and any significant changes that have been made to it.

Chapter 2

General Information

This chapter covers information that is relevant to all the functions specified in Chapter 3 and the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers.

2.1 Use and Implementation of Functions

Each of the following statements shall apply unless explicitly stated otherwise in the detailed descriptions that follow:

1. If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer), the behavior is undefined.

2. Any function declared in a header may also be implemented as a macro defined in the header, so a function should not be declared explicitly if its header is included. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a function even if it is also defined as a macro. The use of the C-language **#undef** construct to remove any such macro definition shall also ensure that an actual function is referred to.

3. Any invocation of a function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments. Likewise, those function-like macros described in the following sections may be invoked in an expression anywhere a function with a compatible return type could be called.

4. Provided that a function can be declared without reference to any type defined in a header, it is also permissible to declare the function, either explicitly or implicitly, and use it without including its associated header.

5. If a function that accepts a variable number of arguments is not declared (explicitly or by including its associated header), the behavior is undefined.

2.2 The Compilation Environment

2.2.1 POSIX.1 Symbols

Certain symbols in this volume of IEEE Std 1003.1-200x are defined in headers (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers). Some of those headers could also define symbols other than those defined by IEEE Std 1003.1-200x, potentially conflicting with symbols used by the application. Also, IEEE Std 1003.1-200x defines symbols that are not permitted by other standards to appear in those headers without some control on the visibility of those symbols.

Symbols called ‘‘feature test macros’’ are used to control the visibility of symbols that might be included in a header. Implementations, future versions of IEEE Std 1003.1-200x, and other standards may define additional feature test macros.

In the compilation of an application that **#define**s a feature test macro specified by IEEE Std 1003.1-200x, no header defined by IEEE Std 1003.1-200x shall be included prior to the definition of the feature test macro. This restriction also applies to any implementation-provided header in which these feature test macros are used. If the definition of the macro does not precede the **#include**, the result is undefined.

Feature test macros shall begin with the underscore character ('_').

2.2.1.1 *The _POSIX_C_SOURCE Feature Test Macro*

A POSIX-conforming application should ensure that the feature test macro _POSIX_C_SOURCE is defined before inclusion of any header.

When an application includes a header described by IEEE Std 1003.1-200x, and when this feature test macro is defined to have the value 200xxxL:

1. All symbols required by IEEE Std 1003.1-200x to appear when the header is included shall be made visible.
2. Symbols that are explicitly permitted, but not required, by IEEE Std 1003.1-200x to appear in that header (including those in reserved name spaces) may be made visible.
3. Additional symbols not required or explicitly permitted by IEEE Std 1003.1-200x to be in that header shall not be made visible, except when enabled by another feature test macro.

Identifiers in IEEE Std 1003.1-200x may only be undefined using the **#undef** directive as described in Section 2.1 (on page 463) or Section 2.2.2. These **#undef** directives shall follow all **#include** directives of any header in IEEE Std 1003.1-200x.

Note: The POSIX.1-1990 standard specified a macro called _POSIX_SOURCE. This has been superseded by _POSIX_C_SOURCE.

2.2.1.2 *The _XOPEN_SOURCE Feature Test Macro*

XSI An XSI-conforming application should ensure that the feature test macro _XOPEN_SOURCE is defined with the value 600 before inclusion of any header. This is needed to enable the functionality described in Section 2.2.1.1 and in addition to enable the X/Open System Interfaces Extension.

Since this volume of IEEE Std 1003.1-200x is aligned with the ISO C standard, and since all functionality enabled by _POSIX_C_SOURCE set equal to 200xxxL is enabled by _XOPEN_SOURCE set equal to 600, there should be no need to define _POSIX_C_SOURCE if _XOPEN_SOURCE is so defined. Therefore, if _XOPEN_SOURCE is set equal to 600 and _POSIX_C_SOURCE is set equal to 200xxxL, the behavior is the same as if only _XOPEN_SOURCE is defined and set equal to 600. However, should _POSIX_C_SOURCE be set to a value greater than 200xxxL, the behavior is unspecified.

2.2.2 The Name Space

All identifiers in this volume of IEEE Std 1003.1-200x, except *environ*, are defined in at least one of the headers, as shown in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers. When XSI _XOPEN_SOURCE or _POSIX_C_SOURCE is defined, each header defines or declares some identifiers, potentially conflicting with identifiers used by the application. The set of identifiers visible to the application consists of precisely those identifiers from the header pages of the included headers, as well as additional identifiers reserved for the implementation. In addition, some headers may make visible identifiers from other headers as indicated on the relevant header pages.

Implementations may also add members to a structure or union without controlling the visibility of those members with a feature test macro, as long as a user-defined macro with the same name cannot interfere with the correct interpretation of the program. The identifiers reserved for use by the implementation are described below:

1. Each identifier with external linkage described in the header section is reserved for use as an identifier with external linkage if the header is included.
2. Each macro described in the header section is reserved for any use if the header is included.
3. Each identifier with file scope described in the header section is reserved for use as an identifier with file scope in the same name space if the header is included.

The prefixes posix_, POSIX_, and _POSIX_ are reserved for use by IEEE Std 1003.1-200x and other POSIX standards. Implementations may add symbols to the headers shown in the following table, provided the identifiers for those symbols begin with the corresponding reserved prefixes in the following table, and do not use the reserved prefixes posix_, POSIX_, or _POSIX_.

	Header	Prefix	Suffix	Complete Name
AIO	**<aio.h>**	aio_, lio_, AIO_, LIO_		
	<arpa/inet.h>	in_, inet_		
	<ctype.h>	to[a-z], is[a-z]		
	<dirent.h>	d_		
	<errno.h>	E[0-9], E[A-Z]		
	<fcntl.h>	l_		
	<glob.h>	gl_		
	<grp.h>	gr_		
	<inttypes.h>			int[0-9a-z_]*_t, uint[0-9a-z_]*_t
	<limits.h>		_MAX, _MIN	
	<locale.h>	LC_[A-Z]		
MSG	**<mqueue.h>**	mq_, MQ_		
XSI	**<ndbm.h>**	dbm_		
	<netdb.h>	h_, n_, p_, s_		
	<net/if.h>	if_		
	<netinet/in.h>	in_, ip_, s_, sin_		
IP6		in6_, s6_, sin6_		
XSI	**<poll.h>**	pd_, ph_, ps_		
	<pthread.h>	pthread_, PTHREAD_		
	<pwd.h>	pw_		
	<regex.h>	re_, rm_		
PS	**<sched.h>**	sched_, SCHED_		
SEM	**<semaphore.h>**	sem_, SEM_		
	<signal.h>	sa_, uc_, SIG[A-Z], SIG_[A-Z]		
XSI		ss_, sv_		
RTS		si_, SI_, sigev_, SIGEV_, sival_		
XSI	**<stropts.h>**	bi_, ic_, l_, sl_, str_		
	<stdint.h>			int[0-9a-z_]*_t, uint[0-9a-z_]*_t
	<stdlib.h>	str[a-z]		
	<string.h>	str[a-z], mem[a-z], wcs[a-z]		
XSI	**<sys/ipc.h>**	ipc_		key, pad, seq
MF	**<sys/mman.h>**	shm_, MAP_, MCL_, MS_, PROT_		
XSI	**<sys/msg.h>**	msg		msg
XSI	**<sys/resource.h>**	rlim_, ru_		
	<sys/select.h>	fd_, fds_, FD_		
XSI	**<sys/sem.h>**	sem		sem
XSI	**<sys/shm.h>**	shm		
	<sys/socket.h>	ss_, sa_, if_, ifc_, ifru_, infu_, ifra_, msg_, cmsg_, l_		
	<sys/stat.h>	st_		
XSI	**<sys/statvfs.h>**	f_		
	<sys/time.h>	fds_, it_, tv_, FD_		
	<sys/times.h>	tms_		

	Header	Prefix	Suffix	Complete Name
XSI	**<sys/uio.h>**	iov_		UIO_MAXIOV
	<sys/un.h>	sun_		
	<sys/utsname.h>	uts_		
XSI	**<sys/wait.h>**	si_, W[A-Z], P_		
	<termios.h>	c_		
	<time.h>	tm_		
TMR		clock_, timer_, it_, tv_,		
TMR		CLOCK_, TIMER_		
XSI	**<ucontext.h>**	uc_, ss_		
XSI	**<ulimit.h>**	UL_		
	<utime.h>	utim_		
XSI	**<utmpx.h>**	ut_	_LVL, _TIME, _PROCESS	
	<wchar.h>	wcs[a-z]		
	<wctype.h>	is[a-z], to[a-z]		
	<wordexp.h>	we_		
	ANY header	POSIX_, _POSIX_, posix_	_t	

Note: The notation [A–Z] indicates any uppercase letter in the portable character set. The notation [a–z] indicates any lowercase letter in the portable character set. Commas and spaces in the lists of prefixes and complete names in the above table are not part of any prefix or complete name.

If any header in the following table is included, macros with the prefixes shown may be defined. After the last inclusion of a given header, an application may use identifiers with the corresponding prefixes for its own purpose, provided their use is preceded by a **#undef** of the corresponding macro.

	Header	Prefix
XSI	**<dlfcn.h>**	RTLD_
	<fcntl.h>	F_, O_, S_
XSI	**<fmtmsg.h>**	MM_
	<fnmatch.h>	FNM_
XSI	**<ftw.h>**	FTW
	<glob.h>	GLOB_
	<inttypes.h>	PRI[a-z], SCN[a-z]
XSI	**<ndbm.h>**	DBM_
	<net/if.h>	IF_
	<netinet/in.h>	IMPLINK_, IN_, INADDR_, IP_, IPPORT_, IPPROTO_, SOCK_
IP6		IPV6_, IN6_
	<netinet/tcp.h>	TCP_
XSI	**<nl_types.h>**	NL_
XSI	**<poll.h>**	POLL
	<regex.h>	REG_
	<signal.h>	SA_, SIG_[0-9a-z_],
XSI		BUS_, CLD_, FPE_, ILL_, POLL_, SEGV_, SI_, SS_, SV_, TRAP_
	stdint.h	INT[0-9A-Z_]_MIN, INT[0-9A-Z_]_MAX, INT[0-9A-Z_]_C UINT[0-9A-Z_]_MIN, UINT[0-9A-Z_]_MAX, UINT[0-9A-Z_]_C
XSI	**<stropts.h>**	FLUSH[A-Z], I_, M_, MUXID_R[A-Z], S_, SND[A-Z], STR
XSI	**<syslog.h>**	LOG_
XSI	**<sys/ipc.h>**	IPC_
XSI	**<sys/mman.h>**	PROT_, MAP_, MS_
XSI	**<sys/msg.h>**	MSG[A-Z]
XSI	**<sys/resource.h>**	PRIO_, RLIM_, RLIMIT_, RUSAGE_
XSI	**<sys/sem.h>**	SEM_
XSI	**<sys/shm.h>**	SHM[A-Z], SHM_[A-Z]
XSI	**<sys/socket.h>**	AF_, CMSG_, MSG_, PF_, SCM_, SHUT_, SO
	<sys/stat.h>	S_
XSI	**<sys/statvfs.h>**	ST_
XSI	**<sys/time.h>**	FD_, ITIMER_
XSI	**<sys/uio.h>**	IOV_
XSI	**<sys/wait.h>**	BUS_, CLD_, FPE_, ILL_, POLL_, SEGV_, SI_, TRAP_
	<termios.h>	V, I, O, TC, B[0-9] (See below.)
	<wordexp.h>	WRDE_

Note: The notation [0–9] indicates any digit. The notation [A–Z] indicates any uppercase letter in the portable character set. The notation [0–9a–z_] indicates any digit, any lowercase letter in the portable character set, or underscore.

The following reserved names are used as exact matches for **<termios.h>**:

XSI

CBAUD	EXTB	VDSUSP
DEFECHO	FLUSHO	VLNEXT
ECHOCTL	LOBLK	VREPRINT
ECHOKE	PENDIN	VSTATUS
ECHOPRT	SWTCH	VWERASE
EXTA	VDISCARD	

The following identifiers are reserved regardless of the inclusion of headers:

1. All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use by the implementation.
2. All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary identifier and tag name spaces.
3. All identifiers in the table below are reserved for use as identifiers with external linkage. Some of these identifiers do not appear in this volume of IEEE Std 1003.1-200x, but are reserved for future use by the ISO C standard.

_Exit	cexp	fesetexceptflag	localtime	scalbn
abort	cexpf	fesetround	log	scalbnf
abs	cexpl	fetestexcept	log10	scalbnl
acos	cimag	feupdateenv	log10f	scanf
acosf	cimagf	fflush	log10l	setbuf
acosh	cimagl	fgetc	log1p	setjmp
acoshf	clearerr	fgetpos	log1pf	setlocale
acoshl	clock	fgets	log1pl	setvbuf
acosl	clog	fgetwc	log2	signal
acosl	clogf	fgetws	log2f	sin
asctime	clogl	floor	log2l	sinf
asin	conj	floorf	logb	sinh
asinf	conjf	floorl	logbf	sinhf
asinh	conjl	fma	logbl	sinhl
asinhf	copysign	fmaf	logf	sinl
asinhl	copysignf	fmal	logl	sprintf
asinl	copysignl	fmax	longjmp	sqrt
asinl	cos	fmaxf	lrint	sqrtf
atan	cosf	fmaxl	lrintf	sqrtl
atan2	cosh	fmin	lrintl	srand
atan2f	coshf	fminf	lround	sscanf
atan2l	coshl	fminl	lroundf	str[a-z]*
atanf	cosl	fmod	lroundl	strtof
atanf	cpow	fmodf	malloc	strtoimax
atanh	cpowf	fmodl	mblen	strtold
atanh	cpowl	fopen	mbrlen	strtoll
atanhf	cproj	fprintf	mbrtowc	strtoull
atanhl	cprojf	fputc	mbsinit	strtoumax
atanl	cprojl	fputs	mbsrtowcs	swprintf
atanl	creal	fputwc	mbstowcs	swscanf
atexit	crealf	fputws	mbtowc	system
atof	creall	fread	mem[a-z]*	tan
atoi	csin	free	mktime	tanf
atol	csinf	freopen	modf	tanh
atoll	csinh	frexp	modff	tanhf
bsearch	csinhf	frexpf	modfl	tanhl
cabs	csinhl	frexpl	nan	tanl
cabsf	csinl	fscanf	nanf	tgamma
cabsl	csqrt	fseek	nanl	tgammaf
cacos	csqrtf	fsetpos	nearbyint	tgammal

cacosf	csqrtl	ftell	nearbyintf	time
cacosh	ctan	fwide	nearbyintl	tmpfile
cacoshf	ctanf	fwprintf	nextafterf	tmpnam
cacoshl	ctanl	fwrite	nextafterl	to[a-z]*
cacosl	ctime	fwscanf	nexttoward	trunc
calloc	difftime	getc	nexttowardf	truncf
carg	div	getchar	nexttowardl	truncl
cargf	erfcf	getenv	perror	ungetc
cargl	erfcl	gets	pow	ungetwc
casin	erff	getwc	powf	va_end
casinf	erfl	getwchar	powl	vfprintf
casinh	errno	gmtime	printf	vfscanf
casinhf	exit	hypotf	putc	vfwprintf
casinhl	exp	hypotl	putchar	vfwscanf
casinl	exp2	ilogb	puts	vprintf
catan	exp2f	ilogbf	putwc	vscanf
catanf	exp2l	ilogbl	putwchar	vsprintf
catanh	expf	imaxabs	qsort	vsscanf
catanh	expl	imaxdiv	raise	vswprintf
catanhf	expm1	is[a-z]*	rand	vswscanf
catanhf	expm1f	isblank	realloc	vwprintf
catanhl	expm1l	iswblank	remainderf	vwscanf
catanhl	fabs	labs	remainderl	wcrtomb
catanl	fabsf	ldexp	remove	wcs[a-z]*
cbrt	fabsl	ldexpf	remquo	wcstof
cbrtf	fclose	ldexpl	remquof	wcstoimax
cbrtl	fdim	ldiv	remquol	wcstold
ccos	fdimf	ldiv	rename	wcstoll
ccosf	fdiml	lgammaf	rewind	wcstoull
ccosh	feclearexcept	lgammal	rint	wcstoumax
ccoshf	fegetenv	llabs	rintf	wctob
ccoshl	fegetexceptflag	llrint	rintl	wctomb
ccosl	fegetround	llrintf	round	wctrans
ceil	feholdexcept	llrintl	roundf	wctype
ceilf	feof	llround	roundl	wcwidth
ceilf	feraiseexcept	llroundf	scalbln	wmem[a-z]*
ceill	ferror	llroundl	scalblnf	wprintf
ceill	fesetenv	localeconv	scalblnl	wscanf

Note: The notation [a–z] indicates any lowercase letter in the portable character set. The notation '*' indicates any combination of digits, letters in the portable character set, or underscore.

4. All functions and external identifiers defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers are reserved for use as identifiers with external linkage.

5. All the identifiers defined in this volume of IEEE Std 1003.1-200x that have external linkage are always reserved for use as identifiers with external linkage.

No other identifiers are reserved.

Applications shall not declare or define identifiers with the same name as an identifier reserved in the same context. Since macro names are replaced whenever found, independent of scope and name space, macro names matching any of the reserved identifier names shall not be defined by

an application if any associated header is included.

Except that the effect of each inclusion of **<assert.h>** depends on the definition of NDEBUG, headers may be included in any order, and each may be included more than once in a given scope, with no difference in effect from that of being included only once.

If used, the application shall ensure that a header is included outside of any external declaration or definition, and it shall be first included before the first reference to any type or macro it defines, or to any function or object it declares. However, if an identifier is declared or defined in more than one header, the second and subsequent associated headers may be included after the initial reference to the identifier. Prior to the inclusion of a header, the application shall not define any macros with names lexically identical to symbols defined by that header.

2.3 Error Numbers

Most functions can provide an error number. The means by which each function provides its error numbers is specified in its description.

Some functions provide the error number in a variable accessed through the symbol *errno*. The symbol *errno*, defined by including the **<errno.h>** header, expands to a modifiable lvalue of type **int**. It is unspecified whether *errno* is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name *errno*, the behavior is undefined.

The value of *errno* should only be examined when it is indicated to be valid by a function's return value. No function in this volume of IEEE Std 1003.1-200x shall set *errno* to zero. For each thread of a process, the value of *errno* shall not be affected by function calls or assignments to *errno* by other threads.

Some functions return an error number directly as the function value. These functions return a value of zero to indicate success.

If more than one error occurs in processing a function call, any one of the possible errors may be returned, as the order of detection is undefined.

Implementations may support additional errors not included in this list, may generate errors included in this list under circumstances other than those described here, or may contain extensions or limitations that prevent some errors from occurring. The ERRORS section on each reference page specifies whether an error shall be returned, or whether it may be returned. Implementations shall not generate a different error number from the ones described here for error conditions described in this volume of IEEE Std 1003.1-200x, but may generate additional errors unless explicitly disallowed for a particular function.

Each implementation shall document, in the conformance document, situations in which each of the optional conditions defined in IEEE Std 1003.1-200x is detected. The conformance document may also contain statements that one or more of the optional error conditions are not detected.

For functions under the Threads option for which [EINTR] is not listed as a possible error condition in this volume of IEEE Std 1003.1-200x, an implementation shall not return an error code of [EINTR].

The following symbolic names identify the possible error numbers, in the context of the functions specifically defined in this volume of IEEE Std 1003.1-200x; these general descriptions are more precisely defined in the ERRORS sections of the functions that return them. Only these symbolic names should be used in programs, since the actual value of the error number is unspecified. All values listed in this section shall be unique integer constant expressions with

836 type **int** suitable for use in **#if** preprocessing directives, except as noted below. The values for all
837 these names shall be found in the **<errno.h>** header defined in the Base Definitions volume of
838 IEEE Std 1003.1-200x. The actual values are unspecified by this volume of IEEE Std 1003.1-200x.

839 [E2BIG]
840 Argument list too long. The sum of the number of bytes used by the new process image's
841 argument list and environment list is greater than the system-imposed limit of {ARG_MAX}
842 bytes.

843 or:

844 Lack of space in an output buffer.

845 or:

846 Argument is greater than the system-imposed maximum.

847 [EACCES]
848 Permission denied. An attempt was made to access a file in a way forbidden by its file
849 access permissions.

850 [EADDRINUSE]
851 Address in use. The specified address is in use.

852 [EADDRNOTAVAIL]
853 Address not available. The specified address is not available from the local system.

854 [EAFNOSUPPORT]
855 Address family not supported. The implementation does not support the specified address
856 family, or the specified address is not a valid address for the address family of the specified
857 socket.

858 [EAGAIN]
859 Resource temporarily unavailable. This is a temporary condition and later calls to the same
860 routine may complete normally.

861 [EALREADY]
862 Connection already in progress. A connection request is already in progress for the specified
863 socket.

864 [EBADF]
865 Bad file descriptor. A file descriptor argument is out of range, refers to no open file, or a
866 read (write) request is made to a file that is only open for writing (reading).

867 [EBADMSG]
868 XSR Bad message. During a *read*(), *getmsg*(), *getpmsg*(), or *ioctl*() I_RECVFD request to a |
869 STREAMS device, a message arrived at the head of the STREAM that is inappropriate for
870 the function receiving the message.

871 *read*() Message waiting to be read on a STREAM is not a data message. |

872 *getmsg*() or *getpmsg*() |
873 A file descriptor was received instead of a control message. |

874 *ioctl*() Control or data information was received instead of a file descriptor when
875 I_RECVFD was specified.

876 or:

877 Bad Message. The implementation has detected a corrupted message.

878 [EBUSY]
879 Resource busy. An attempt was made to make use of a system resource that is not currently
880 available, as it is being used by another process in a manner that would have conflicted with
881 the request being made by this process.

882 [ECANCELED]
883 Operation canceled. The associated asynchronous operation was canceled before
884 completion.

885 [ECHILD]
886 No child process. A *wait*( ) or *waitpid*( ) function was executed by a process that had no
887 existing or unwaited-for child process.

888 [ECONNABORTED]
889 Connection aborted. The connection has been aborted.

890 [ECONNREFUSED]
891 Connection refused. An attempt to connect to a socket was refused because there was no
892 process listening or because the queue of connection requests was full and the underlying
893 protocol does not support retransmissions.

894 [ECONNRESET]
895 Connection reset. The connection was forcibly closed by the peer.

896 [EDEADLK]
897 Resource deadlock would occur. An attempt was made to lock a system resource that
898 would have resulted in a deadlock situation.

899 [EDESTADDRREQ]
900 Destination address required. No bind address was established.

901 [EDOM]
902 Domain error. An input argument is outside the defined domain of the mathematical
903 function (defined in the ISO C standard).

904 [EDQUOT]
905 Reserved.

906 [EEXIST]
907 File exists. An existing file was mentioned in an inappropriate context; for example, as a
908 new link name in the *link*( ) function.

909 [EFAULT]
910 Bad address. The system detected an invalid address in attempting to use an argument of a
911 call. The reliable detection of this error cannot be guaranteed, and when not detected may
912 result in the generation of a signal, indicating an address violation, which is sent to the
913 process.

914 [EFBIG]
915 File too large. The size of a file would exceed the maximum file size of an implementation or
916 offset maximum established in the corresponding file description.

917 [EHOSTUNREACH]
918 Host is unreachable. The destination host cannot be reached (probably because the host is
919 down or a remote router cannot reach it).

920 [EIDRM]
921 Identifier removed. Returned during XSI interprocess communication if an identifier has
922 been removed from the system.

[EILSEQ]
Illegal byte sequence. A wide-character code has been detected that does not correspond to a valid character, or a byte sequence does not form a valid wide-character code (defined in the ISO C standard).

[EINPROGRESS]
Operation in progress. This code is used to indicate that an asynchronous operation has not yet completed.

or:

O_NONBLOCK is set for the socket file descriptor and the connection cannot be immediately established.

[EINTR]
Interrupted function call. An asynchronous signal was caught by the process during the execution of an interruptible function. If the signal handler performs a normal return, the interrupted function call may return this condition (see the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**).

[EINVAL]
Invalid argument. Some invalid argument was supplied; for example, specifying an undefined signal in a *signal*( ) function or a *kill*( ) function.

[EIO]
Input/output error. Some physical input or output error has occurred. This error may be reported on a subsequent operation on the same file descriptor. Any other error-causing operation on the same file descriptor may cause the [EIO] error indication to be lost.

[EISCONN]
Socket is connected. The specified socket is already connected.

[EISDIR]
Is a directory. An attempt was made to open a directory with write mode specified.

[ELOOP]
Symbolic link loop. A loop exists in symbolic links encountered during pathname resolution. This error may also be returned if more than {SYMLOOP_MAX} symbolic links are encountered during pathname resolution.

[EMFILE]
Too many open files. An attempt was made to open more than the maximum number of {OPEN_MAX} file descriptors allowed in this process.

[EMLINK]
Too many links. An attempt was made to have the link count of a single file exceed {LINK_MAX}.

[EMSGSIZE]
Message too large. A message sent on a transport provider was larger than an internal message buffer or some other network limit.

or:

Inappropriate message buffer length.

[EMULTIHOP]
Reserved.

[ENAMETOOLONG]
Filename too long. The length of a pathname exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX}. This error may also occur when pathname substitution, as a result of encountering a symbolic link during pathname resolution, results in a pathname string the size of which exceeds {PATH_MAX}.

[ENETDOWN]
Network is down. The local network interface used to reach the destination is down.

[ENETRESET]
The connection was aborted by the network.

[ENETUNREACH]
Network unreachable. No route to the network is present.

[ENFILE]
Too many files open in system. Too many files are currently open in the system. The system has reached its predefined limit for simultaneously open files and temporarily cannot accept requests to open another one.

[ENOBUFS]
No buffer space available. Insufficient buffer resources were available in the system to perform the socket operation.

XSR [ENODATA]
No message available. No message is available on the STREAM head read queue.

[ENODEV]
No such device. An attempt was made to apply an inappropriate function to a device; for example, trying to read a write-only device such as a printer.

[ENOENT]
No such file or directory. A component of a specified pathname does not exist, or the pathname is an empty string.

[ENOEXEC]
Executable file format error. A request is made to execute a file that, although it has the appropriate permissions, is not in the format required by the implementation for executable files.

[ENOLCK]
No locks available. A system-imposed limit on the number of simultaneous file and record locks has been reached and no more are currently available.

[ENOLINK]
Reserved.

[ENOMEM]
Not enough space. The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.

[ENOMSG]
No message of the desired type. The message queue does not contain a message of the required type during XSI interprocess communication.

[ENOPROTOOPT]
Protocol not available. The protocol option specified to *setsockopt*( ) is not supported by the implementation.

[ENOSPC]
No space left on a device. During the *write*( ) function on a regular file or when extending a directory, there is no free space left on the device.

XSR [ENOSR]
No STREAM resources. Insufficient STREAMS memory resources are available to perform a STREAMS-related function. This is a temporary condition; it may be recovered from if other processes release resources.

XSR [ENOSTR]
Not a STREAM. A STREAM function was attempted on a file descriptor that was not associated with a STREAMS device.

[ENOSYS]
Function not implemented. An attempt was made to use a function that is not available in this implementation.

[ENOTCONN]
Socket not connected. The socket is not connected.

[ENOTDIR]
Not a directory. A component of the specified pathname exists, but it is not a directory, when a directory was expected.

[ENOTEMPTY]
Directory not empty. A directory other than an empty directory was supplied when an empty directory was expected.

[ENOTSOCK]
Not a socket. The file descriptor does not refer to a socket.

[ENOTSUP]
Not supported. The implementation does not support this feature of the Realtime Option Group.

[ENOTTY]
Inappropriate I/O control operation. A control function has been attempted for a file or special file for which the operation is inappropriate.

[ENXIO]
No such device or address. Input or output on a special file refers to a device that does not exist, or makes a request beyond the capabilities of the device. It may also occur when, for example, a tape drive is not on-line.

[EOPNOTSUPP]
Operation not supported on socket. The type of socket (address family or protocol) does not support the requested operation.

[EOVERFLOW]
Value too large to be stored in data type. An operation was attempted which would generate a value that is outside the range of values that can be represented in the relevant data type or that are allowed for a given data item.

[EPERM]
Operation not permitted. An attempt was made to perform an operation limited to processes with appropriate privileges or to the owner of a file or other resource.

[EPIPE]
Broken pipe. A write was attempted on a socket, pipe, or FIFO for which there is no process

to read the data.

[EPROTO]
Protocol error. Some protocol error occurred. This error is device-specific, but is generally not related to a hardware failure.

[EPROTONOSUPPORT]
Protocol not supported. The protocol is not supported by the address family, or the protocol is not supported by the implementation.

[EPROTOTYPE]
Protocol wrong type for socket. The socket type is not supported by the protocol.

[ERANGE]
Result too large or too small. The result of the function is too large (overflow) or too small (underflow) to be represented in the available space (defined in the ISO C standard).

[EROFS]
Read-only file system. An attempt was made to modify a file or directory on a file system that is read-only.

[ESPIPE]
Invalid seek. An attempt was made to access the file offset associated with a pipe or FIFO.

[ESRCH]
No such process. No process can be found corresponding to that specified by the given process ID.

[ESTALE]
Reserved.

XSR [ETIME]
STREAM *ioctl*( ) timeout. The timer set for a STREAMS *ioctl*( ) call has expired. The cause of this error is device-specific and could indicate either a hardware or software failure, or a timeout value that is too short for the specific operation. The status of the *ioctl*( ) operation is unspecified.

[ETIMEDOUT]
Connection timed out. The connection to a remote machine has timed out. If the connection timed out during execution of the function that reported this error (as opposed to timing out prior to the function being called), it is unspecified whether the function has completed some or all of the documented behavior associated with a successful completion of the function.

or:

Operation timed out. The time limit associated with the operation was exceeded before the operation completed.

[ETXTBSY]
Text file busy. An attempt was made to execute a pure-procedure program that is currently open for writing, or an attempt has been made to open for writing a pure-procedure program that is being executed.

[EWOULDBLOCK]
Operation would block. An operation on a socket marked as non-blocking has encountered a situation such as no data available that otherwise would have caused the function to suspend execution.

A conforming implementation may assign the same values for [EWOULDBLOCK] and [EAGAIN].

[EXDEV]
Improper link. A link to a file on another file system was attempted.

2.3.1 Additional Error Numbers

Additional implementation-defined error numbers may be defined in **<errno.h>**.

2.4 Signal Concepts

2.4.1 Signal Generation and Delivery

A signal is said to be *generated* for (or sent to) a process or thread when the event that causes the signal first occurs. Examples of such events include detection of hardware faults, timer expiration, RTS signals generated via the **sigevent** structure RTS and terminal activity, as well as invocations of the *kill*( ) and *sigqueue*( ) functions. In some circumstances, the same event generates signals for multiple processes.

At the time of generation, a determination shall be made whether the signal has been generated for the process or for a specific thread within the process. Signals which are generated by some action attributable to a particular thread, such as a hardware fault, shall be generated for the thread that caused the signal to be generated. Signals that are generated in association with a process ID or process group ID or an asynchronous event, such as terminal activity, shall be generated for the process.

Each process has an action to be taken in response to each signal defined by the system (see Section 2.4.3 (on page 480)). A signal is said to be *delivered* to a process when the appropriate action for the process and signal is taken. A signal is said to be *accepted* by a process when the signal is selected and returned by one of the *sigwait*( ) functions.

During the time between the generation of a signal and its delivery or acceptance, the signal is said to be *pending*. Ordinarily, this interval cannot be detected by an application. However, a signal can be *blocked* from delivery to a thread. If the action associated with a blocked signal is anything other than to ignore the signal, and if that signal is generated for the thread, the signal shall remain pending until it is unblocked, it is accepted when it is selected and returned by a call to the *sigwait*( ) function, or the action associated with it is set to ignore the signal. Signals generated for the process shall be delivered to exactly one of those threads within the process which is in a call to a *sigwait*( ) function selecting that signal or has not blocked delivery of the signal. If there are no threads in a call to a *sigwait*( ) function selecting that signal, and if all threads within the process block delivery of the signal, the signal shall remain pending on the process until a thread calls a *sigwait*( ) function selecting that signal, a thread unblocks delivery of the signal, or the action associated with the signal is set to ignore the signal. If the action associated with a blocked signal is to ignore the signal and if that signal is generated for the process, it is unspecified whether the signal is discarded immediately upon generation or remains pending.

Each thread has a *signal mask* that defines the set of signals currently blocked from delivery to it. | The signal mask for a thread shall be initialized from that of its parent or creating thread, or from | the corresponding thread in the parent process if the thread was created as the result of a call to | *fork*( ). The *sigaction*( ), *sigprocmask*( ), and *sigsuspend*( ) functions control the manipulation of the signal mask.

The determination of which action is taken in response to a signal is made at the time the signal is delivered, allowing for any changes since the time of generation. This determination is independent of the means by which the signal was originally generated. If a subsequent occurrence of a pending signal is generated, it is implementation-defined as to whether the
RTS signal is delivered or accepted more than once in circumstances other than those in which queuing is required under the Realtime Signals Extension option. The order in which multiple, simultaneously pending signals outside the range SIGRTMIN to SIGRTMAX are delivered to or accepted by a process is unspecified.

When any stop signal (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) is generated for a process, any pending SIGCONT signals for that process shall be discarded. Conversely, when SIGCONT is generated for a process, all pending stop signals for that process shall be discarded. When SIGCONT is generated for a process that is stopped, the process shall be continued, even if the SIGCONT signal is blocked or ignored. If SIGCONT is blocked and not ignored, it shall remain pending until it is either unblocked or a stop signal is generated for the process.

An implementation shall document any condition not specified by this volume of IEEE Std 1003.1-200x under which the implementation generates signals.

2.4.2 Realtime Signal Generation and Delivery

RTS This section describes extensions to support realtime signal generation and delivery. This functionality is dependent on support of the Realtime Signals Extension option (and the rest of this section is not further shaded for this option).

Some signal-generating functions, such as high-resolution timer expiration, asynchronous I/O completion, interprocess message arrival, and the *sigqueue*( ) function, support the specification of an application-defined value, either explicitly as a parameter to the function or in a **sigevent** structure parameter. The **sigevent** structure is defined in **<signal.h>** and contains at least the following members:

Member Type	Member Name	Description
int	*sigev_notify*	Notification type.
int	*sigev_signo*	Signal number.
union sigval	*sigev_value*	Signal value.
void(*)(unsigned sigval)	*sigev_notify_function*	Notification function.
(pthread_attr_t*)	*sigev_notify_attributes*	Notification attributes.

The *sigev_notify* member specifies the notification mechanism to use when an asynchronous event occurs. This volume of IEEE Std 1003.1-200x defines the following values for the *sigev_notify* member:

SIGEV_NONE No asynchronous notification shall be delivered when the event of interest occurs.

SIGEV_SIGNAL The signal specified in *sigev_signo* shall be generated for the process when the event of interest occurs. If the implementation supports the Realtime Signals Extension option and if the SA_SIGINFO flag is set for that signal number, then the signal shall be queued to the process and the value specified in *sigev_value* shall be the *si_value* component of the generated signal. If SA_SIGINFO is not set for that signal number, it is unspecified whether the signal is queued and what value, if any, is sent.

SIGEV_THREAD A notification function shall be called to perform notification.

An implementation may define additional notification mechanisms.

The *sigev_signo* member specifies the signal to be generated. The *sigev_value* member is the application-defined value to be passed to the signal-catching function at the time of the signal delivery or to be returned at signal acceptance as the *si_value* member of the **siginfo_t** structure.

The **sigval** union is defined in **<signal.h>** and contains at least the following members:

Member Type	Member Name	Description
int	*sival_int*	Integer signal value.
void*	*sival_ptr*	Pointer signal value.

The *sival_int* member shall be used when the application-defined value is of type **int**; the *sival_ptr* member shall be used when the application-defined value is a pointer.

When a signal is generated by the *sigqueue*( ) function or any signal-generating function that supports the specification of an application-defined value, the signal shall be marked pending and, if the SA_SIGINFO flag is set for that signal, the signal shall be queued to the process along with the application-specified signal value. Multiple occurrences of signals so generated are queued in FIFO order. It is unspecified whether signals so generated are queued when the SA_SIGINFO flag is not set for that signal.

Signals generated by the *kill*( ) function or other events that cause signals to occur, such as detection of hardware faults, *alarm*( ) timer expiration, or terminal activity, and for which the implementation does not support queuing, shall have no effect on signals already queued for the same signal number.

When multiple unblocked signals, all in the range SIGRTMIN to SIGRTMAX, are pending, the behavior shall be as if the implementation delivers the pending unblocked signal with the lowest signal number within that range. No other ordering of signal delivery is specified.

If, when a pending signal is delivered, there are additional signals queued to that signal number, the signal shall remain pending. Otherwise, the pending indication shall be reset.

Multi-threaded programs can use an alternate event notification mechanism. When a notification is processed, and the *sigev_notify* member of the **sigevent** structure has the value SIGEV_THREAD, the function *sigev_notify_function* is called with parameter *sigev_value*.

The function shall be executed in an environment as if it were the *start_routine* for a newly created thread with thread attributes specified by *sigev_notify_attributes*. If *sigev_notify_attributes* is NULL, the behavior shall be as if the thread were created with the *detachstate* attribute set to PTHREAD_CREATE_DETACHED. Supplying an attributes structure with a *detachstate* attribute of PTHREAD_CREATE_JOINABLE results in undefined behavior. The signal mask of this thread is implementation-defined.

2.4.3 Signal Actions

There are three types of action that can be associated with a signal: SIG_DFL, SIG_IGN, or a pointer to a function. Initially, all signals shall be set to SIG_DFL or SIG_IGN prior to entry of the *main*( ) routine (see the *exec* functions). The actions prescribed by these values are as follows:

SIG_DFL Signal-specific default action.

RTS The default actions for the signals defined in this volume of IEEE Std 1003.1-200x are specified under **<signal.h>**. If the Realtime Signals Extension option is supported, the default actions for the realtime signals in the range SIGRTMIN to SIGRTMAX shall be to terminate the process abnormally.

If the default action is to stop the process, the execution of that process is temporarily suspended. When a process stops, a SIGCHLD signal shall be generated for its parent process, unless the parent process has set the SA_NOCLDSTOP flag. While a process is stopped, any additional signals that are sent to the process shall not be delivered until the process is continued, except SIGKILL which always terminates the receiving process. A process that is a member of an orphaned process group shall not be allowed to stop in response to the SIGTSTP, SIGTTIN, or SIGTTOU signals. In cases where delivery of one of these signals would stop such a process, the signal shall be discarded.

Setting a signal action to SIG_DFL for a signal that is pending, and whose default action is to ignore the signal (for example, SIGCHLD), shall cause the pending signal to be discarded, whether or not it is blocked.

The default action for SIGCONT is to resume execution at the point where the RTS process was stopped, after first handling any pending unblocked signals. If the Realtime Signals Extension option is supported, any queued values pending shall be discarded and the resources used to queue them shall be released and returned to the system for other use. When a stopped process is continued, a SIGCHLD signal may be generated for its parent process, unless the parent process has set the SA_NOCLDSTOP flag.

SIG_IGN Ignore signal.

Delivery of the signal shall have no effect on the process. The behavior of a process RTS is undefined after it ignores a SIGFPE, SIGILL, SIGSEGV, or SIGBUS signal that RTS was not generated by *kill*( ), *sigqueue*( ), or *raise*( ).

The system shall not allow the action for the signals SIGKILL or SIGSTOP to be set to SIG_IGN.

Setting a signal action to SIG_IGN for a signal that is pending shall cause the pending signal to be discarded, whether or not it is blocked.

If a process sets the action for the SIGCHLD signal to SIG_IGN, the behavior is XSI unspecified, except as specified below.

If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the calling processes shall not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited-for children that were transformed into zombie processes, it shall block until all of its children terminate, and *wait*( ), *waitid*( ), and *waitpid*( ) shall fail and set *errno* to [ECHILD].

RTS If the Realtime Signals Extension option is supported, any queued values pending shall be discarded and the resources used to queue them shall be released and made available to queue other signals.

pointer to a function
Catch signal.

On delivery of the signal, the receiving process is to execute the signal-catching function at the specified address. After returning from the signal-catching function, the receiving process shall resume execution at the point at which it was interrupted.

If the SA_SIGINFO flag for the signal is cleared, the signal-catching function shall be entered as a C-language function call as follows:

```
void func(int signo);
```

XSI | RTS If the SA_SIGINFO flag for the signal is set, the signal-catching function shall be entered as a C-language function call as follows:

```
void func(int signo, siginfo_t *info, void *context);
```

where *func* is the specified signal-catching function, *signo* is the signal number of the signal being delivered, and *info* is a pointer to a **siginfo_t** structure defined in **<signal.h>** containing at least the following members:

Member Type	Member Name	Description
int	*si_signo*	Signal number.
int	*si_code*	Cause of the signal.
union sigval	*si_value*	Signal value.

The *si_signo* member shall contain the signal number. This shall be the same as the *signo* parameter. The *si_code* member shall contain a code identifying the cause of the signal. The following values are defined for *si_code*:

SI_USER The signal was sent by the *kill*( ) function. The implementation may set *si_code* to SI_USER if the signal was sent by the *raise*( ) or *abort*( ) functions or any similar functions provided as implementation extensions.

RTS SI_QUEUE The signal was sent by the *sigqueue*( ) function.

RTS SI_TIMER The signal was generated by the expiration of a timer set by *timer_settime*( ).

RTS SI_ASYNCIO The signal was generated by the completion of an asynchronous I/O request.

RTS SI_MESGQ The signal was generated by the arrival of a message on an empty message queue.

If the signal was not generated by one of the functions or events listed above, the *si_code* shall be set to an implementation-defined value that is not equal to any of the values defined above.

RTS If the Realtime Signals Extension is supported, and *si_code* is one of SI_QUEUE, SI_TIMER, SI_ASYNCIO, or SI_MESGQ, then *si_value* shall contain the application-specified signal value. Otherwise, the contents of *si_value* are undefined.

The behavior of a process is undefined after it returns normally from a signal-
XSI catching function for a SIGBUS, SIGFPE, SIGILL, or SIGSEGV signal that was not
RTS generated by *kill*( ), *sigqueue*( ), or *raise*( ).

The system shall not allow a process to catch the signals SIGKILL and SIGSTOP.

If a process establishes a signal-catching function for the SIGCHLD signal while it has a terminated child process for which it has not waited, it is unspecified whether a SIGCHLD signal is generated to indicate that child process.

When signal-catching functions are invoked asynchronously with process execution, the behavior of some of the functions defined by this volume of IEEE Std 1003.1-200x is unspecified if they are called from a signal-catching function.

The following table defines a set of functions that shall be either reentrant or non-interruptible by signals and shall be async-signal-safe. Therefore applications may invoke them, without restriction, from signal-catching functions:

Notes to Reviewers

This section with side shading will not appear in the final copy. - Ed.

The contents of the following tables need to be reviewed.

_Exit( )	*fdatasync*( )	*posix_trace_event*( )	*sigsuspend*( )
_exit( )	*fork*( )	*raise*( )	*stat*( )
access( )	*fpathconf*( )	*read*( )	*symlink*( )
aio_error( )	*fstat*( )	*readlink*( )	*sysconf*( )
aio_return( )	*fsync*( )	*rename*( )	*tcdrain*( )
aio_suspend( )	*ftruncate*( )	*rmdir*( )	*tcflow*( )
alarm( )	*getegid*( )	*sem_post*( )	*tcflush*( )
cfgetispeed( )	*geteuid*( )	*setgid*( )	*tcgetattr*( )
cfgetospeed( )	*getgid*( )	*setpgid*( )	*tcgetpgrp*( )
cfsetispeed( )	*getgroups*( )	*setsid*( )	*tcsendbreak*( )
cfsetospeed( )	*getpgrp*( )	*setuid*( )	*tcsetattr*( )
chdir( )	*getpid*( )	*sigaction*( )	*tcsetpgrp*( )
chmod( )	*getppid*( )	*sigaddset*( )	*time*( )
chown( )	*getuid*( )	*sigdelset*( )	*timer_getoverrun*( )
clock_gettime( )	*kill*( )	*sigemptyset*( )	*timer_gettime*( )
close( )	*link*( )	*sigfillset*( )	*timer_settime*( )
creat( )	*lseek*( )	*sigismember*( )	*times*( )
dup( )	*lstat*( )	*sleep*( )	*umask*( )
dup2( )	*mkdir*( )	*signal*( )	*uname*( )
execle( )	*mkfifo*( )	*sigpause*( )	*unlink*( )
execve( )	*open*( )	*sigpending*( )	*utime*( )
fchmod( )	*pathconf*( )	*sigprocmask*( )	*wait*( )
fchown( )	*pause*( )	*sigqueue*( )	*waitpid*( )
fcntl( )	*pipe*( )	*sigset*( )	*write*( )

All functions not in the above table are considered to be unsafe with respect to signals. In the presence of signals, all functions defined by this volume of IEEE Std 1003.1-200x shall behave as defined when called from or interrupted by a signal-catching function, with a single exception: when a signal interrupts an unsafe function and the signal-catching function calls an unsafe function, the behavior is undefined.

When a signal is delivered to a thread, if the action of that signal specifies termination, stop, or continue, the entire process shall be terminated, stopped, or continued, respectively.

2.4.4 Signal Effects on Other Functions

Signals affect the behavior of certain functions defined by this volume of IEEE Std 1003.1-200x if delivered to a process while it is executing such a function. If the action of the signal is to terminate the process, the process shall be terminated and the function shall not return. If the action of the signal is to stop the process, the process shall stop until continued or terminated. Generation of a SIGCONT signal for the process shall cause the process to be continued, and the original function shall continue at the point the process was stopped. If the action of the signal is to invoke a signal-catching function, the signal-catching function shall be invoked; in this case the original function is said to be *interrupted* by the signal. If the signal-catching function executes a **return** statement, the behavior of the interrupted function shall be as described individually for that function, except as noted for unsafe functions. Signals that are ignored shall not affect the behavior of any function; signals that are blocked shall not affect the behavior of any function until they are unblocked and then delivered, except as specified for the *sigpending*( ) and *sigwait*( ) functions.

2.5 Standard I/O Streams

A stream is associated with an external file (which may be a physical device) by *opening* a file, which may involve *creating* a new file. Creating an existing file causes its former contents to be discarded if necessary. If a file can support positioning requests, (such as a disk file, as opposed to a terminal), then a *file position indicator* associated with the stream is positioned at the start (byte number 0) of the file, unless the file is opened with append mode, in which case it is implementation-defined whether the file position indicator is initially positioned at the beginning or end of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file. All input takes place as if bytes were read by successive calls to *fgetc*( ); all output takes place as if bytes were written by successive calls to *fputc*( ).

When a stream is *unbuffered*, bytes are intended to appear from the source or at the destination as soon as possible; otherwise, bytes may be accumulated and transmitted as a block. When a stream is *fully buffered*, bytes are intended to be transmitted as a block when a buffer is filled. When a stream is *line buffered*, bytes are intended to be transmitted as a block when a newline byte is encountered. Furthermore, bytes are intended to be transmitted as a block when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line-buffered stream that requires the transmission of bytes. Support for these characteristics is implementation-defined, and may be affected via *setbuf*( ) and *setvbuf*( ).

A file may be disassociated from a controlling stream by *closing* the file. Output streams are flushed (any unwritten buffer contents are transmitted) before the stream is disassociated from the file. The value of a pointer to a **FILE** object is unspecified after the associated file is closed (including the standard streams).

A file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the *main*( ) function returns to its original caller, or if the *exit*( ) function is called, all open files are closed (hence all output streams are flushed) before program termination. Other paths to program termination, such as calling *abort*( ), need not close all files properly.

The address of the **FILE** object used to control a stream may be significant; a copy of a **FILE** object need not necessarily serve in place of the original.

At program start-up, three streams are predefined and need not be opened explicitly: *standard input* (for reading conventional input), *standard output* (for writing conventional output), and

standard error (for writing diagnostic output). When opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

2.5.1 Interaction of File Descriptors and Standard I/O Streams

CX This section describes the interaction of file descriptors and standard I/O streams. This functionality is an extension to the ISO C standard (and the rest of this section is not further CX shaded).

An open file description may be accessed through a file descriptor, which is created using functions such as *open*( ) or *pipe*( ), or through a stream, which is created using functions such as *fopen*( ) or *popen*( ). Either a file descriptor or a stream is called a *handle* on the open file description to which it refers; an open file description may have several handles.

Handles can be created or destroyed by explicit user action, without affecting the underlying open file description. Some of the ways to create them include *fcntl*( ), *dup*( ), *fdopen*( ), *fileno*( ), and *fork*( ). They can be destroyed by at least *fclose*( ), *close*( ), and the *exec* functions.

A file descriptor that is never used in an operation that could affect the file offset (for example, *read*( ), *write*( ), or *lseek*( )) is not considered a handle for this discussion, but could give rise to one (for example, as a consequence of *fdopen*( ), *dup*( ), or *fork*( )). This exception does not include the file descriptor underlying a stream, whether created with *fopen*( ) or *fdopen*( ), so long as it is not used directly by the application to affect the file offset. The *read*( ) and *write*( ) functions implicitly affect the file offset; *lseek*( ) explicitly affects it.

The result of function calls involving any one handle (the *active handle*) is defined elsewhere in this volume of IEEE Std 1003.1-200x, but if two or more handles are used, and any one of them is a stream, the application shall ensure that their actions are coordinated as described below. If this is not done, the result is undefined.

A handle which is a stream is considered to be closed when either an *fclose*( ) or *freopen*( ) is executed on it (the result of *freopen*( ) is a new stream, which cannot be a handle on the same open file description as its previous value), or when the process owning that stream terminates with *exit*( ) or *abort*( ). A file descriptor is closed by *close*( ), *_exit*( ), or the *exec* functions when FD_CLOEXEC is set on that file descriptor.

For a handle to become the active handle, the application shall ensure that the actions below are performed between the last use of the handle (the current active handle) and the first use of the second handle (the future active handle). The second handle then becomes the active handle. All activity by the application affecting the file offset on the first handle shall be suspended until it again becomes the active file handle. (If a stream function has as an underlying function one that affects the file offset, the stream function shall be considered to affect the file offset.)

The handles need not be in the same process for these rules to apply.

Note that after a *fork*( ), two handles exist where one existed before. The application shall ensure that, if both handles can ever be accessed, they are both in a state where the other could become the active handle first. The application shall prepare for a *fork*( ) exactly as if it were a change of active handle. (If the only action performed by one of the processes is one of the *exec* functions or *_exit*( ) (not *exit*( )), the handle is never accessed in that process.)

For the first handle, the first applicable condition below applies. After the actions required below are taken, if the handle is still open, the application can close it.

- If it is a file descriptor, no action is required.

- If the only further action to be performed on any handle to this open file descriptor is to close it, no action need be taken.
- If it is a stream which is unbuffered, no action need be taken.
- If it is a stream which is line buffered, and the last byte written to the stream was a newline (that is, as if a:

  ```
  putc('\n')
  ```

  was the most recent operation on that stream), no action need be taken.
- If it is a stream which is open for writing or appending (but not also open for reading), the application shall either perform an *fflush*( ), or the stream shall be closed.
- If the stream is open for reading and it is at the end of the file (*feof*( ) is true), no action need be taken.
- If the stream is open with a mode that allows reading and the underlying open file description refers to a device that is capable of seeking, the application shall either perform an *fflush*( ), or the stream shall be closed.

Otherwise, the result is undefined.

For the second handle:

- If any previous active handle has been used by a function that explicitly changed the file offset, except as required above for the first handle, the application shall perform an *lseek*( ) or *fseek*( ) (as appropriate to the type of handle) to an appropriate location.

If the active handle ceases to be accessible before the requirements on the first handle, above, have been met, the state of the open file description becomes undefined. This might occur during functions such as a *fork*( ) or *_exit*( ).

The *exec* functions make inaccessible all streams that are open at the time they are called, independent of which streams or file descriptors may be available to the new process image.

When these rules are followed, regardless of the sequence of handles used, implementations shall ensure that an application, even one consisting of several processes, shall yield correct results: no data shall be lost or duplicated when writing, and all data shall be written in order, except as requested by seeks. It is implementation-defined whether, and under what conditions, all input is seen exactly once.

If the rules above are not followed, the result is unspecified.

Each function that operates on a stream is said to have zero or more *underlying functions*. This means that the stream function shares certain traits with the underlying functions, but does not require that there be any relation between the implementations of the stream function and its underlying functions.

2.5.2 Stream Orientation and Encoding Rules

For conformance to the ISO/IEC 9899:1999 standard, the definition of a stream includes an *orientation*. After a stream is associated with an external file, but before any operations are performed on it, the stream is without orientation. Once a wide-character input/output function has been applied to a stream without orientation, the stream shall become *wide-oriented*. Similarly, once a byte input/output function has been applied to a stream without orientation, the stream shall become *byte-oriented*. Only a call to the *freopen*( ) function or the *fwide*( ) function can otherwise alter the orientation of a stream.

A successful call to *freopen*( ) shall remove any orientation. The three predefined streams *standard input*, *standard output*, and *standard error* shall be unoriented at program start-up.

Byte input/output functions cannot be applied to a wide-oriented stream, and wide-character input/output functions cannot be applied to a byte-oriented stream. The remaining stream operations shall not affect and shall not be affected by a stream's orientation, except for the following additional restrictions:

- Binary wide-oriented streams have the file positioning restrictions ascribed to both text and binary streams.
- For wide-oriented streams, after a successful call to a file-positioning function that leaves the file position indicator prior to the end-of-file, a wide-character output function can overwrite a partial character; any file contents beyond the byte(s) written are henceforth undefined.

Each wide-oriented stream has an associated **mbstate_t** object that stores the current parse state of the stream. A successful call to *fgetpos*( ) shall store a representation of the value of this **mbstate_t** object as part of the value of the **fpos_t** object. A later successful call to *fsetpos*( ) using the same stored **fpos_t** value shall restore the value of the associated **mbstate_t** object as well as the position within the controlled stream.

Implementations that support multiple encoding rules associate an encoding rule with the stream. The encoding rule shall be determined by the setting of the *LC_CTYPE* category in the current locale at the time when the stream becomes wide-oriented. If a wide-character input/output function is applied to a byte-oriented stream, the encoding rule used is undefined. As with the stream's orientation, the encoding rule associated with a stream cannot be changed once it has been set, except by a successful call to *freopen*( ) which clears the encoding rule and resets the orientation to unoriented.

Although both text and binary wide-oriented streams are conceptually sequences of wide characters, the external file associated with a wide-oriented stream is a sequence of (possibly multi-byte) characters generalized as follows:

- Multi-byte encodings within files may contain embedded null bytes (unlike multi-byte encodings valid for use internal to the program).
- A file need not begin nor end in the initial shift state.

Moreover, the encodings used for characters may differ among files. Both the nature and choice of such encodings are implementation-defined.

The wide-character input functions read characters from the stream and convert them to wide characters as if they were read by successive calls to the *fgetwc*( ) function. Each conversion shall occur as if by a call to the *mbrtowc*( ) function, with the conversion state described by the stream's own **mbstate_t** object, CX except the encoding rule associated with the stream is used instead of the encoding rule implied by the *LC_CTYPE* category of the current locale.

The wide-character output functions convert wide characters to (possibly multi-byte) characters and write them to the stream as if they were written by successive calls to the *fputwc*( ) function. Each conversion shall occur as if by a call to the *wcrtomb*( ) function, with the conversion state described by the stream's own **mbstate_t** object, CX except the encoding rule associated with the stream is used instead of the encoding rule implied by the *LC_CTYPE* category of the current locale.

An *encoding error* shall occur if the character sequence presented to the underlying *mbrtowc*( ) function does not form a valid (generalized) character, or if the code value passed to the underlying *wcrtomb*( ) function does not correspond to a valid (generalized) character. The wide-character input/output functions and the byte input/output functions store the value of

the macro [EILSEQ] in *errno* if and only if an encoding error occurs.

2.6 STREAMS

XSR STREAMS functionality is provided on implementations supporting the XSI STREAMS Option Group. This functionality is dependent on support of the XSI STREAMS option (and the rest of this section is not further shaded for this option).

STREAMS provides a uniform mechanism for implementing networking services and other character-based I/O. The STREAMS function provides direct access to protocol modules. STREAMS modules are unspecified objects. Access to STREAMS modules is provided by interfaces in IEEE Std 1003.1-200x. Creation of STREAMS modules is outside the scope of IEEE Std 1003.1-200x.

A STREAM is typically a full-duplex connection between a process and an open device or pseudo-device. However, since pipes may be STREAMS-based, a STREAM can be a full-duplex connection between two processes. The STREAM itself exists entirely within the implementation and provides a general character I/O function for processes. It optionally includes one or more intermediate processing modules that are interposed between the process end of the STREAM (STREAM head) and a device driver at the end of the STREAM (STREAM end).

STREAMS I/O is based on messages. There are three types of message:

- *Data messages* containing actual data for input or output
- *Control data* containing instructions for the STREAMS modules and underlying implementation
- Other messages, which include file descriptors

The interface between the STREAM and the rest of the implementation is provided by a set of functions at the STREAM head. When a process calls *write*(), *writev*(), *putmsg*(), *putpmsg*(), or *ioctl*(), messages are sent down the STREAM, and *read*(), *readv*(), *getmsg*(), or *getpmsg*() accepts data from the STREAM and passes it to a process. Data intended for the device at the downstream end of the STREAM is packaged into messages and sent downstream, while data and signals from the device are composed into messages by the device driver and sent upstream to the STREAM head.

When a STREAMS-based device is opened, a STREAM shall be created that contains the STREAM head and the STREAM end (driver). If pipes are STREAMS-based in an implementation, when a pipe is created, two STREAMS shall be created, each containing a STREAM head. Other modules are added to the STREAM using *ioctl*(). New modules are ''pushed'' onto the STREAM one at a time in last-in, first-out (LIFO) style, as though the STREAM was a push-down stack.

Priority

Message types are classified according to their queuing priority and may be *normal* (non-priority), *priority*, or *high-priority* messages. A message belongs to a particular priority band that determines its ordering when placed on a queue. Normal messages have a priority band of 0 and shall always be placed at the end of the queue following all other messages in the queue. High-priority messages are always placed at the head of a queue, but shall be discarded if there is already a high-priority message in the queue. Their priority band shall be ignored; they are high-priority by virtue of their type. Priority messages have a priority band greater than 0. Priority messages are always placed after any messages of the same or higher priority. High-priority and priority messages are used to send control and data information outside the normal

flow of control. By convention, high-priority messages shall not be affected by flow control. Normal and priority messages have separate flow controls.

Message Parts

A process may access STREAMS messages that contain a data part, control part, or both. The data part is that information which is transmitted over the communication medium and the control information is used by the local STREAMS modules. The other types of messages are used between modules and are not accessible to processes. Messages containing only a data part are accessible via *putmsg*( ), *putpmsg*( ), *getmsg*( ), *getpmsg*( ), *read*( ), *readv*( ), *write*( ), or *writev*( ). Messages containing a control part with or without a data part are accessible via calls to *putmsg*( ), *putpmsg*( ), *getmsg*( ), or *getpmsg*( ).

2.6.1 Accessing STREAMS

A process accesses STREAMS-based files using the standard functions *close*( ), *ioctl*( ), *getmsg*( ), *getpmsg*( ), *open*( ), *pipe*( ), *poll*( ), *putmsg*( ), *putpmsg*( ), *read*( ), or *write*( ). Refer to the applicable function definitions for general properties and errors.

Calls to *ioctl*( ) shall perform control functions on the STREAM associated with the file descriptor *fildes*. The control functions may be performed by the STREAM head, a STREAMS module, or the STREAMS driver for the STREAM.

STREAMS modules and drivers can detect errors, sending an error message to the STREAM head, thus causing subsequent functions to fail and set *errno* to the value specified in the message. In addition, STREAMS modules and drivers can elect to fail a particular *ioctl*( ) request alone by sending a negative acknowledgement message to the STREAM head. This shall cause just the pending *ioctl*( ) request to fail and set *errno* to the value specified in the message.

2.7 XSI Interprocess Communication

XSI This section describes extensions to support interprocess communication. This functionality is dependent on support of the XSI Extension (and the rest of this section is not further shaded for this option).

The following message passing, semaphore, and shared memory services form an XSI interprocess communication facility. Certain aspects of their operation are common, and are described below.

IPC Functions		
msgctl( )	*semctl*( )	*shmctl*( )
msgget( )	*semget*( )	*shmdt*( )
msgrcv( )	*semop*( )	*shmget*( )
msgsnd( )	*shmat*( )	

Another interprocess communication facility is provided by functions in the Realtime Option Group; see Section 2.8 (on page 491).

2.7.1 IPC General Description

Each individual shared memory segment, message queue, and semaphore set shall be identified by a unique positive integer, called, respectively, a shared memory identifier, *shmid*, a semaphore identifier, *semid*, and a message queue identifier, *msqid*. The identifiers shall be returned by calls to *shmget*( ), *semget*( ), and *msgget*( ), respectively.

Associated with each identifier is a data structure which contains data related to the operations which may be or may have been performed; see the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/shm.h>**, **<sys/sem.h>**, and **<sys/msg.h>** for their descriptions.

Each of the data structures contains both ownership information and an **ipc_perm** structure (see the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/ipc.h>**) which are used in conjunction to determine whether or not read/write (read/alter for semaphores) permissions should be granted to processes using the IPC facilities. The *mode* member of the **ipc_perm** structure acts as a bit field which determines the permissions.

The values of the bits are given below in octal notation.

Bit	Meaning
0400	Read by user.
0200	Write by user.
0040	Read by group.
0020	Write by group.
0004	Read by others.
0002	Write by others.

The name of the **ipc_perm** structure is *shm_perm*, *sem_perm*, or *msg_perm*, depending on which service is being used. In each case, read and write/alter permissions shall be granted to a process if one or more of the following are true ("xxx" is replaced by *shm*, *sem*, or *msg*, as appropriate):

- The process has appropriate privileges.
- The effective user ID of the process matches *xxx_perm.cuid* or *xxx_perm.uid* in the data structure associated with the IPC identifier, and the appropriate bit of the *user* field in *xxx_perm.mode* is set.
- The effective user ID of the process does not match *xxx_perm.cuid* or *xxx_perm.uid* but the effective group ID of the process matches *xxx_perm.cgid* or *xxx_perm.gid* in the data structure associated with the IPC identifier, and the appropriate bit of the *group* field in *xxx_perm.mode* is set.
- The effective user ID of the process does not match *xxx_perm.cuid* or *xxx_perm.uid* and the effective group ID of the process does not match *xxx_perm.cgid* or *xxx_perm.gid* in the data structure associated with the IPC identifier, but the appropriate bit of the *other* field in *xxx_perm.mode* is set.

Otherwise, the permission shall be denied.

2.8 Realtime

This section defines functions to support the source portability of applications with realtime requirements. The presence of many of these functions is dependent on support for implementation options described in the text.

The specific functional areas included in this section and their scope include the following. Full definitions of these terms can be found in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 3, Definitions.

- Semaphores
- Process Memory Locking
- Memory Mapped Files and Shared Memory Objects
- Priority Scheduling
- Realtime Signal Extension
- Timers
- Interprocess Communication
- Synchronized Input and Output
- Asynchronous Input and Output

All the realtime functions defined in this volume of IEEE Std 1003.1-200x are portable, although some of the numeric parameters used by an implementation may have hardware dependencies.

2.8.1 Realtime Signals

RTS Realtime signal generation and delivery is dependent on support for the Realtime Signals Extension option.

See Section 2.4.2 (on page 479).

2.8.2 Asynchronous I/O

AIO The functionality described in this section is dependent on support of the Asynchronous Input and Output option (and the rest of this section is not further shaded for this option).

An asynchronous I/O control block structure **aiocb** is used in many asynchronous I/O functions. It is defined in the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>** and has at least the following members:

Member Type	Member Name	Description
int	*aio_fildes*	File descriptor.
off_t	*aio_offset*	File offset.
volatile void*	*aio_buf*	Location of buffer.
size_t	*aio_nbytes*	Length of transfer.
int	*aio_reqprio*	Request priority offset.
struct sigevent	*aio_sigevent*	Signal number and value.
int	*aio_lio_opcode*	Operation to be performed.

The *aio_fildes* element is the file descriptor on which the asynchronous operation is performed.

If O_APPEND is not set for the file descriptor *aio_fildes* and if *aio_fildes* is associated with a device that is capable of seeking, then the requested operation takes place at the absolute position in the file as given by *aio_offset*, as if *lseek*( ) were called immediately prior to the

operation with an *offset* argument equal to *aio_offset* and a *whence* argument equal to SEEK_SET. If O_APPEND is set for the file descriptor, or if *aio_fildes* is associated with a device that is incapable of seeking, write operations append to the file in the same order as the calls were made, with the following exception: under implementation-defined circumstances, such as operation on a multi-processor or when requests of differing priorities are submitted at the same time, the ordering restriction may be relaxed. Since there is no way for a strictly conforming application to determine whether this relaxation applies, all strictly conforming applications which rely on ordering of output shall be written in such a way that they will operate correctly if the relaxation applies. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified. The *aio_nbytes* and *aio_buf* elements are the same as the *nbyte* and *buf* arguments defined by *read*( ) and *write*( ), respectively.

If _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING are defined, then asynchronous I/O is queued in priority order, with the priority of each asynchronous operation based on the current scheduling priority of the calling process. The *aio_reqprio* member can be used to lower (but not raise) the asynchronous I/O operation priority and is within the range zero through {AIO_PRIO_DELTA_MAX}, inclusive. Unless both _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING are defined, the order of processing asynchronous I/O requests is unspecified. When both _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING are defined, the order of processing of requests submitted by processes whose schedulers are not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC is unspecified. The priority of an asynchronous request is computed as (process scheduling priority) minus *aio_reqprio*. The priority assigned to each asynchronous I/O request is an indication of the desired order of execution of the request relative to other asynchronous I/O requests for this file. If _POSIX_PRIORITIZED_IO is defined, requests issued with the same priority to a character special file are processed by the underlying device in FIFO order; the order of processing of requests of the same priority issued to files that are not character special files is unspecified. Numerically higher priority values indicate requests of higher priority. The value of *aio_reqprio* has no effect on process scheduling priority. When prioritized asynchronous I/O requests to the same file are blocked waiting for a resource required for that I/O operation, the higher-priority I/O requests shall be granted the resource before lower-priority I/O requests are granted the resource. The relative priority of asynchronous I/O and synchronous I/O is implementation-defined. If _POSIX_PRIORITIZED_IO is defined, the implementation shall define for which files I/O prioritization is supported.

The *aio_sigevent* determines how the calling process shall be notified upon I/O completion, as specified in Section 2.4.1 (on page 478). If *aio_sigevent.sigev_notify* is SIGEV_NONE, then no signal shall be posted upon I/O completion, but the error status for the operation and the return status for the operation shall be set appropriately.

The *aio_lio_opcode* field is used only by the *lio_listio*( ) call. The *lio_listio*( ) call allows multiple asynchronous I/O operations to be submitted at a single time. The function takes as an argument an array of pointers to **aiocb** structures. Each **aiocb** structure indicates the operation to be performed (read or write) via the *aio_lio_opcode* field.

The address of the **aiocb** structure is used as a handle for retrieving the error status and return status of the asynchronous operation while it is in progress.

The **aiocb** structure and the data buffers associated with the asynchronous I/O operation are being used by the system for asynchronous I/O while, and only while, the error status of the asynchronous operation is equal to [EINPROGRESS]. Applications shall not modify the **aiocb** structure while the structure is being used by the system for asynchronous I/O.

The return status of the asynchronous operation is the number of bytes transferred by the I/O operation. If the error status is set to indicate an error completion, then the return status is set to

the return value that the corresponding *read*(), *write*(), or *fsync*() call would have returned. When the error status is not equal to [EINPROGRESS], the return status shall reflect the return status of the corresponding synchronous operation.

2.8.3 Memory Management

2.8.3.1 Memory Locking

ML The functionality described in this section is dependent on support of the Process Memory Locking option (and the rest of this section is not further shaded for this option).

Range memory locking operations are defined in terms of pages. Implementations may restrict the size and alignment of range lockings to be on page-size boundaries. The page size, in bytes, is the value of the configurable system variable {PAGESIZE}. If an implementation has no restrictions on size or alignment, it may specify a 1-byte page size.

Memory locking guarantees the residence of portions of the address space. It is implementation-defined whether locking memory guarantees fixed translation between virtual addresses (as seen by the process) and physical addresses. Per-process memory locks are not inherited across a *fork*(), and all memory locks owned by a process are unlocked upon *exec* or process termination. Unmapping of an address range removes any memory locks established on that address range by this process.

2.8.3.2 Memory Mapped Files

MF The functionality described in this section is dependent on support of the Memory Mapped Files option (and the rest of this section is not further shaded for this option).

Range memory mapping operations are defined in terms of pages. Implementations may restrict the size and alignment of range mappings to be on page-size boundaries. The page size, in bytes, is the value of the configurable system variable {PAGESIZE}. If an implementation has no restrictions on size or alignment, it may specify a 1-byte page size.

Memory mapped files provide a mechanism that allows a process to access files by directly incorporating file data into its address space. Once a file is mapped into a process address space, the data can be manipulated as memory. If more than one process maps a file, its contents are shared among them. If the mappings allow shared write access, then data written into the memory object through the address space of one process appears in the address spaces of all processes that similarly map the same portion of the memory object.

SHM Shared memory objects are named regions of storage that may be independent of the file system and can be mapped into the address space of one or more processes to allow them to share the associated memory.

SHM An *unlink*() of a file or *shm_unlink*() of a shared memory object, while causing the removal of the name, does not unmap any mappings established for the object. Once the name has been removed, the contents of the memory object are preserved as long as it is referenced. The memory object remains referenced as long as a process has the memory object open or has some area of the memory object mapped.

2.8.3.3 *Memory Protection*

MPR MF The functionality described in this section is dependent on support of the Memory Protection and Memory Mapped Files option (and the rest of this section is not further shaded for these options).

When an object is mapped, various application accesses to the mapped region may result in signals. In this context, SIGBUS is used to indicate an error using the mapped object, and SIGSEGV is used to indicate a protection violation or misuse of an address:

- A mapping may be restricted to disallow some types of access.
- Write attempts to memory that was mapped without write access, or any access to memory mapped PROT_NONE, shall result in a SIGSEGV signal.
- References to unmapped addresses shall result in a SIGSEGV signal.
- Reference to whole pages within the mapping, but beyond the current length of the object, shall result in a SIGBUS signal.
- The size of the object is unaffected by access beyond the end of the object (even if a SIGBUS is not generated).

2.8.3.4 *Typed Memory Objects*

TYM The functionality described in this section is dependent on support of the Typed Memory Objects option (and the rest of this section is not further shaded for this option).

Implementations may support the Typed Memory Objects option without supporting the Memory Mapped Files option or the Shared Memory Objects option. Typed memory objects are implementation-configurable named storage pools accessible from one or more processors in a system, each via one or more ports, such as backplane buses, LANs, I/O channels, and so on. Each valid combination of a storage pool and a port is identified through a name that is defined at system configuration time, in an implementation-defined manner; the name may be independent of the file system. Using this name, a typed memory object can be opened and mapped into process address space. For a given storage pool and port, it is necessary to support both dynamic allocation from the pool as well as mapping at an application-supplied offset within the pool; when dynamic allocation has been performed, subsequent deallocation must be supported. Lastly, accessing typed memory objects from different ports requires a method for obtaining the offset and length of contiguous storage of a region of typed memory (dynamically allocated or not); this allows typed memory to be shared among processes and/or processors while being accessed from the desired port.

2.8.4 Process Scheduling

PS The functionality described in this section is dependent on support of the Process Scheduling option (and the rest of this section is not further shaded for this option).

Scheduling Policies

The scheduling semantics described in this volume of IEEE Std 1003.1-200x are defined in terms of a conceptual model that contains a set of thread lists. No implementation structures are necessarily implied by the use of this conceptual model. It is assumed that no time elapses during operations described using this model, and therefore no simultaneous operations are possible. This model discusses only processor scheduling for runnable threads, but it should be noted that greatly enhanced predictability of realtime applications results if the sequencing of other resources takes processor scheduling policy into account.

There is, conceptually, one thread list for each priority. A runnable thread will be on the thread list for that thread's priority. Multiple scheduling policies shall be provided. Each non-empty thread list is ordered, contains a head as one end of its order, and a tail as the other. The purpose of a scheduling policy is to define the allowable operations on this set of lists (for example, moving threads between and within lists).

Each process shall be controlled by an associated scheduling policy and priority. These parameters may be specified by explicit application execution of the *sched_setscheduler*( ) or *sched_setparam*( ) functions.

Each thread shall be controlled by an associated scheduling policy and priority. These parameters may be specified by explicit application execution of the *pthread_setschedparam*( ) function.

Associated with each policy is a priority range. Each policy definition shall specify the minimum priority range for that policy. The priority ranges for each policy may but need not overlap the priority ranges of other policies.

A conforming implementation shall select the thread that is defined as being at the head of the highest priority non-empty thread list to become a running thread, regardless of its associated policy. This thread is then removed from its thread list.

Four scheduling policies are specifically required. Other implementation-defined scheduling policies may be defined. The following symbols are defined in the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**:

SCHED_FIFO	First in, first out (FIFO) scheduling policy.
SCHED_RR	Round robin scheduling policy.
SS SCHED_SPORADIC	Sporadic server scheduling policy.
SCHED_OTHER	Another scheduling policy.

The values of these symbols shall be distinct.

SCHED_FIFO

Conforming implementations shall include a scheduling policy called the FIFO scheduling policy.

Threads scheduled under this policy are chosen from a thread list that is ordered by the time its threads have been on the list without being executed; generally, the head of the list is the thread that has been on the list the longest time, and the tail is the thread that has been on the list the shortest time.

Under the SCHED_FIFO policy, the modification of the definitional thread lists is as follows:

1. When a running thread becomes a preempted thread, it becomes the head of the thread list for its priority.
2. When a blocked thread becomes a runnable thread, it becomes the tail of the thread list for its priority.
3. When a running thread calls the *sched_setscheduler*( ) function, the process specified in the function call is modified to the specified policy and the priority specified by the *param* argument.
4. When a running thread calls the *sched_setparam*( ) function, the priority of the process specified in the function call is modified to the priority specified by the *param* argument.

5. When a running thread calls the *pthread_setschedparam*( ) function, the thread specified in the function call is modified to the specified policy and the priority specified by the *param* argument.
6. When a running thread calls the *pthread_setschedprio*( ) function, the thread specified in the function call is modified to the priority specified by the *prio* argument.
7. If a thread whose policy or priority has been modified other than by *pthread_setschedprio*( ) is a running thread or is runnable, it then becomes the tail of the thread list for its new priority.
8. If a thread whose policy or priority has been modified by *pthread_setschedprio*( ) is a running thread or is runnable, the effect on its position in the thread list depends on the direction of the modification, as follows:
    a. If the priority is raised, the thread becomes the tail of the thread list.
    b. If the priority is unchanged, the thread does not change position in the thread list.
    c. If the priority is lowered, the thread becomes the head of the thread list.
9. When a running thread issues the *sched_yield*( ) function, the thread becomes the tail of the thread list for its priority.
10. At no other time is the position of a thread with this scheduling policy within the thread lists affected.

For this policy, valid priorities shall be within the range returned by the *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions when SCHED_FIFO is provided as the parameter. Conforming implementations shall provide a priority range of at least 32 priorities for this policy.

SCHED_RR

Conforming implementations shall include a scheduling policy called the *round robin* scheduling policy. This policy shall be identical to the SCHED_FIFO policy with the additional condition that when the implementation detects that a running thread has been executing as a running thread for a time period of the length returned by the *sched_rr_get_interval*( ) function or longer, the thread shall become the tail of its thread list and the head of that thread list shall be removed and made a running thread.

The effect of this policy is to ensure that if there are multiple SCHED_RR threads at the same priority, one of them does not monopolize the processor. An application should not rely only on the use of SCHED_RR to ensure application progress among multiple threads if the application includes threads using the SCHED_FIFO policy at the same or higher priority levels or SCHED_RR threads at a higher priority level.

A thread under this policy that is preempted and subsequently resumes execution as a running thread completes the unexpired portion of its round robin interval time period.

For this policy, valid priorities shall be within the range returned by the *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions when SCHED_RR is provided as the parameter. Conforming implementations shall provide a priority range of at least 32 priorities for this policy.

SCHED_SPORADIC

SS | TSP The functionality described in this section is dependent on support of the Process Sporadic Server or Thread Sporadic Server options (and the rest of this section is not further shaded for these options).

If _POSIX_SPORADIC_SERVER or _POSIX_THREAD_SPORADIC_SERVER is defined, the implementation shall include a scheduling policy identified by the value SCHED_SPORADIC.

The sporadic server policy is based primarily on two parameters: the *replenishment period* and the *available execution capacity*. The replenishment period is given by the *sched_ss_repl_period* member of the **sched_param** structure. The available execution capacity is initialized to the value given by the *sched_ss_init_budget* member of the same parameter. The sporadic server policy is identical to the SCHED_FIFO policy with some additional conditions that cause the thread's assigned priority to be switched between the values specified by the *sched_priority* and *sched_ss_low_priority* members of the **sched_param** structure.

The priority assigned to a thread using the sporadic server scheduling policy is determined in the following manner: if the available execution capacity is greater than zero and the number of pending replenishment operations is strictly less than *sched_ss_max_repl*, the thread is assigned the priority specified by *sched_priority*; otherwise, the assigned priority shall be *sched_ss_low_priority*. If the value of *sched_priority* is less than or equal to the value of *sched_ss_low_priority*, the results are undefined. When active, the thread shall belong to the thread list corresponding to its assigned priority level, according to the mentioned priority assignment. The modification of the available execution capacity and, consequently of the assigned priority, is done as follows:

1. When the thread at the head of the *sched_priority* list becomes a running thread, its execution time shall be limited to at most its available execution capacity, plus the resolution of the execution time clock used for this scheduling policy. This resolution shall be implementation-defined.

2. Each time the thread is inserted at the tail of the list associated with *sched_priority*—because as a blocked thread it became runnable with priority *sched_priority* or because a replenishment operation was performed—the time at which this operation is done is posted as the *activation_time*.

3. When the running thread with assigned priority equal to *sched_priority* becomes a preempted thread, it becomes the head of the thread list for its priority, and the execution time consumed is subtracted from the available execution capacity. If the available execution capacity would become negative by this operation, it shall be set to zero.

4. When the running thread with assigned priority equal to *sched_priority* becomes a blocked thread, the execution time consumed is subtracted from the available execution capacity, and a replenishment operation is scheduled, as described in 6 and 7. If the available execution capacity would become negative by this operation, it shall be set to zero.

5. When the running thread with assigned priority equal to *sched_priority* reaches the limit imposed on its execution time, it becomes the tail of the thread list for *sched_ss_low_priority*, the execution time consumed is subtracted from the available execution capacity (which becomes zero), and a replenishment operation is scheduled, as described in 6 and 7.

6. Each time a replenishment operation is scheduled, the amount of execution capacity to be replenished, *replenish_amount*, is set equal to the execution time consumed by the thread since the *activation_time*. The replenishment is scheduled to occur at *activation_time* plus *sched_ss_repl_period*. If the scheduled time obtained is before the current time, the

replenishment operation is carried out immediately. Several replenishment operations may be pending at the same time, each of which will be serviced at its respective scheduled time. With the above rules, the number of replenishment operations simultaneously pending for a given thread that is scheduled under the sporadic server policy shall not be greater than *sched_ss_max_repl*.

7. A replenishment operation consists of adding the corresponding *replenish_amount* to the available execution capacity at the scheduled time. If, as a consequence of this operation, the execution capacity would become larger than *sched_ss_initial_budget*, it shall be rounded down to a value equal to *sched_ss_initial_budget*. Additionally, if the thread was runnable or running, and had assigned priority equal to *sched_ss_low_priority*, then it becomes the tail of the thread list for *sched_priority*.

Execution time is defined in Section 2.2.2 (on page 464).

For this policy, changing the value of a CPU-time clock via *clock_settime*( ) shall have no effect on its behavior.

For this policy, valid priorities shall be within the range returned by the *sched_get_priority_min*( ) and *sched_get_priority_max*( ) functions when SCHED_SPORADIC is provided as the parameter. Conforming implementations shall provide a priority range of at least 32 distinct priorities for this policy.

SCHED_OTHER

Conforming implementations shall include one scheduling policy identified as SCHED_OTHER (which may execute identically with either the FIFO or round robin scheduling policy). The effect of scheduling threads with the SCHED_OTHER policy in a system in which other threads SS are executing under SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC is implementation-defined.

This policy is defined to allow strictly conforming applications to be able to indicate in a portable manner that they no longer need a realtime scheduling policy.

For threads executing under this policy, the implementation shall use only priorities within the range returned by the *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions when SCHED_OTHER is provided as the parameter.

2.8.5 Clocks and Timers

TMR The functionality described in this section is dependent on support of the Timers option (and the rest of this section is not further shaded for this option).

The **<time.h>** header defines the types and manifest constants used by the timing facility.

Time Value Specification Structures

Many of the timing facility functions accept or return time value specifications. A time value structure **timespec** specifies a single time value and includes at least the following members:

Member Type	Member Name	Description
time_t	*tv_sec*	Seconds.
long	*tv_nsec*	Nanoseconds.

The *tv_nsec* member is only valid if greater than or equal to zero, and less than the number of nanoseconds in a second (1,000 million). The time interval described by this structure is (*tv_sec* * 10^9 + *tv_nsec*) nanoseconds.

A time value structure **itimerspec** specifies an initial timer value and a repetition interval for use by the per-process timer functions. This structure includes at least the following members:

Member Type	Member Name	Description
struct timespec	*it_interval*	Timer period.
struct timespec	*it_value*	Timer expiration.

If the value described by *it_value* is non-zero, it indicates the time to or time of the next timer expiration (for relative and absolute timer values, respectively). If the value described by *it_value* is zero, the timer shall be disarmed.

If the value described by *it_interval* is non-zero, it specifies an interval which shall be used in reloading the timer when it expires; that is, a periodic timer is specified. If the value described by *it_interval* is zero, the timer is disarmed after its next expiration; that is, a one-shot timer is specified.

Timer Event Notification Control Block

RTS Per-process timers may be created that notify the process of timer expirations by queuing a realtime extended signal. The **sigevent** structure, defined in the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, is used in creating such a timer. The **sigevent** structure contains the signal number and an application-specific data value which shall be used when notifying the calling process of timer expiration events.

Manifest Constants

The following constants are defined in the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**:

CLOCK_REALTIME — The identifier for the system-wide realtime clock.

TIMER_ABSTIME — Flag indicating time is absolute with respect to the clock associated with a timer.

MON CLOCK_MONOTONIC — The identifier for the system-wide monotonic clock, which is defined as a clock whose value cannot be set via *clock_settime*( ) and which cannot have backward clock jumps. The maximum possible clock jump is implementation-defined.

MON The maximum allowable resolution for CLOCK_REALTIME and CLOCK_MONOTONIC clocks and all time services based on these clocks is represented by {_POSIX_CLOCKRES_MIN} and shall be defined as 20ms (1/50 of a second). Implementations may support smaller values of resolution for these clocks to provide finer granularity time bases. The actual resolution supported by an implementation for a specific clock is obtained using the *clock_getres*( ) function. If the actual resolution supported for a time service based on one of these clocks differs from the resolution supported for that clock, the implementation shall document this difference.

MON The minimum allowable maximum value for CLOCK_REALTIME and CLOCK_MONOTONIC clocks and all absolute time services based on them is the same as that defined by the ISO C standard for the **time_t** type. If the maximum value supported by a time service based on one of these clocks differs from the maximum value supported by that clock, the implementation shall document this difference.

Execution Time Monitoring

CPT If _POSIX_CPUTIME is defined, process CPU-time clocks shall be supported in addition to the clocks described in **Manifest Constants** (on page 499).

TCT If _POSIX_THREAD_CPUTIME is defined, thread CPU-time clocks shall be supported.

CPT|TCT CPU-time clocks measure execution or CPU time, which is defined in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.117, CPU Time (Execution Time). The mechanism used to measure execution time is described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.9, Measurement of Execution Time.

CPT If _POSIX_CPUTIME is defined, the following constant of the type **clockid_t** is defined in **<time.h>**:

CLOCK_PROCESS_CPUTIME_ID
When this value of the type **clockid_t** is used in a *clock*() or *timer**() function call, it is interpreted as the identifier of the CPU-time clock associated with the process making the function call.

TCT If _POSIX_THREAD_CPUTIME is defined, the following constant of the type **clockid_t** is defined in **<time.h>**:

CLOCK_THREAD_CPUTIME_ID
When this value of the type **clockid_t** is used in a *clock*() or *timer**() function call, it is interpreted as the identifier of the CPU-time clock associated with the thread making the function call.

2.9 Threads

THR The functionality described in this section is dependent on support of the Threads option (and the rest of this section is not further shaded for this option).

This section defines functionality to support multiple flows of control, called *threads*, within a process. For the definition of *threads*, see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.393, Thread.

The specific functional areas covered by threads and their scope include:

- Thread management: the creation, control, and termination of multiple flows of control in the same process under the assumption of a common shared address space
- Synchronization primitives optimized for tightly coupled operation of multiple control flows in a common, shared address space

2.9.1 Thread-Safety

All functions defined by this volume of IEEE Std 1003.1-200x shall be thread-safe, except that the following functions[1] need not be thread-safe.

---

1. The functions in the table are not shaded to denote applicable options. Individual reference pages should be consulted.

asctime( )	*ecvt*( )	*gethostent*( )	*getutxline*( )	*putenv*( )	
basename( )	*encrypt*( )	*getlogin*( )	*gmtime*( )	*pututxline*( )	
catgets( )	*endgrent*( )	*getnetbyaddr*( )	*hcreate*( )	*rand*( )	
crypt( )	*endpwent*( )	*getnetbyname*( )	*hdestroy*( )	*readdir*( )	
ctime( )	*endutxent*( )	*getnetent*( )	*hsearch*( )	*setenv*( )	
dbm_clearerr( )	*fcvt*( )	*getopt*( )	*inet_ntoa*( )	*setgrent*( )	
dbm_close( )	*ftw*( )	*getprotobyname*( )	*l64a*( )	*setkey*( )	
dbm_delete( )	*gcvt*( )	*getprotobynumber*( )	*lgamma*( )	*setpwent*( )	
dbm_error( )	*getc_unlocked*( )	*getprotoent*( )	*localeconv*( )	*setutxent*( )	
dbm_fetch( )	*getchar_unlocked*( )	*getpwent*( )	*localtime*( )	*strerror*( )	
dbm_firstkey( )	*getdate*( )	*getpwnam*( )	*lrand48*( )	*strtok*( )	
dbm_nextkey( )	*getenv*( )	*getpwuid*( )	*mrand48*( )	*ttyname*( )	
dbm_open( )	*getgrent*( )	*getservbyname*( )	*nftw*( )	*unsetenv*( )	
dbm_store( )	*getgrgid*( )	*getservbyport*( )	*nl_langinfo*( )	*wcstombs*( )	
dirname( )	*getgrnam*( )	*getservent*( )	*ptsname*( )	*wctomb*( )	
dlerror( )	*gethostbyaddr*( )	*getutxent*( )	*putc_unlocked*( )		
drand48( )	*gethostbyname*( )	*getutxid*( )	*putchar_unlocked*( )		

The *ctermid*( ) and *tmpnam*( ) functions need not be thread-safe if passed a NULL argument. The *wcrtomb*( ) and *wcsrtombs*( ) functions need not be thread-safe if passed a NULL *ps* argument.

Implementations shall provide internal synchronization as necessary in order to satisfy this requirement.

2.9.2 Thread IDs

Although implementations may have thread IDs that are unique in a system, applications should only assume that thread IDs are usable and unique within a single process. The effect of calling any of the functions defined in this volume of IEEE Std 1003.1-200x and passing as an argument the thread ID of a thread from another process is unspecified. A conforming implementation is free to reuse a thread ID after the thread terminates if it was created with the *detachstate* attribute set to PTHREAD_CREATE_DETACHED or if *pthread_detach*( ) or *pthread_join*( ) has been called for that thread. If a thread is detached, its thread ID is invalid for use as an argument in a call to *pthread_detach*( ) or *pthread_join*( ).

2.9.3 Thread Mutexes

A thread that has blocked shall not prevent any unblocked thread that is eligible to use the same processing resources from eventually making forward progress in its execution. Eligibility for processing resources is determined by the scheduling policy.

A thread shall become the owner of a mutex, *m*, when one of the following occurs:

- It returns successfully from *pthread_mutex_lock*( ) with *m* as the *mutex* argument.
- It returns successfully from *pthread_mutex_trylock*( ) with *m* as the *mutex* argument.
- TMO It returns successfully from *pthread_mutex_timedwait*( ) with *m* as the mutex argument.
- It returns (successfully or not) from *pthread_cond_wait*( ) with *m* as the *mutex* argument (except as explicitly indicated otherwise for certain errors).
- It returns (successfully or not) from *pthread_cond_timedwait*( ) with *m* as the *mutex* argument (except as explicitly indicated otherwise for certain errors).

The thread shall remain the owner of *m* until one of the following occurs:

- It executes *pthread_mutex_unlock*( ) with *m* as the *mutex* argument
- It blocks in a call to *pthread_cond_wait*( ) with *m* as the *mutex* argument.
- It blocks in a call to *pthread_cond_timedwait*( ) with *m* as the *mutex* argument.

The implementation shall behave as if at all times there is at most one owner of any mutex.

A thread that becomes the owner of a mutex is said to have *acquired* the mutex and the mutex is said to have become *locked*; when a thread gives up ownership of a mutex it is said to have *released* the mutex and the mutex is said to have become *unlocked*.

2.9.4 Thread Scheduling

TPS The functionality described in this section is dependent on support of the Thread Execution Scheduling option (and the rest of this section is not further shaded for this option).

Thread Scheduling Attributes

In support of the scheduling function, threads have attributes which are accessed through the **pthread_attr_t** thread creation attributes object.

The *contentionscope* attribute defines the scheduling contention scope of the thread to be either PTHREAD_SCOPE_PROCESS or PTHREAD_SCOPE_SYSTEM.

The *inheritsched* attribute specifies whether a newly created thread is to inherit the scheduling attributes of the creating thread or to have its scheduling values set according to the other scheduling attributes in the **pthread_attr_t** object.

The *schedpolicy* attribute defines the scheduling policy for the thread. The *schedparam* attribute defines the scheduling parameters for the thread. The interaction of threads having different policies within a process is described as part of the definition of those policies.

If the Thread Execution Scheduling option is defined, and the *schedpolicy* attribute specifies one of the priority-based policies defined under this option, the *schedparam* attribute contains the scheduling priority of the thread. A conforming implementation ensures that the priority value in *schedparam* is in the range associated with the scheduling policy when the thread attributes object is used to create a thread, or when the scheduling attributes of a thread are dynamically modified. The meaning of the priority value in *schedparam* is the same as that of *priority*.

TSP If _POSIX_THREAD_SPORADIC_SERVER is defined, the *schedparam* attribute supports four new members that are used for the sporadic server scheduling policy. These members are *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, and *sched_ss_max_repl*. The meaning of these attributes is the same as in the definitions that appear under Section 2.8.4 (on page 494).

When a process is created, its single thread has a scheduling policy and associated attributes equal to the process' policy and attributes. The default scheduling contention scope value is implementation-defined. The default values of other scheduling attributes are implementation-defined.

Thread Scheduling Contention Scope

The scheduling contention scope of a thread defines the set of threads with which the thread competes for use of the processing resources. The scheduling operation selects at most one thread to execute on each processor at any point in time and the thread's scheduling attributes (for example, *priority*), whether under process scheduling contention scope or system scheduling contention scope, are the parameters used to determine the scheduling decision.

The scheduling contention scope, in the context of scheduling a mixed scope environment, affects threads as follows:

- A thread created with PTHREAD_SCOPE_SYSTEM scheduling contention scope contends for resources with all other threads in the same scheduling allocation domain relative to their system scheduling attributes. The system scheduling attributes of a thread created with PTHREAD_SCOPE_SYSTEM scheduling contention scope are the scheduling attributes with which the thread was created. The system scheduling attributes of a thread created with PTHREAD_SCOPE_PROCESS scheduling contention scope are the implementation-defined mapping into system attribute space of the scheduling attributes with which the thread was created.
- Threads created with PTHREAD_SCOPE_PROCESS scheduling contention scope contend directly with other threads within their process that were created with PTHREAD_SCOPE_PROCESS scheduling contention scope. The contention is resolved based on the threads' scheduling attributes and policies. It is unspecified how such threads are scheduled relative to threads in other processes or threads with PTHREAD_SCOPE_SYSTEM scheduling contention scope.
- Conforming implementations shall support the PTHREAD_SCOPE_PROCESS scheduling contention scope, the PTHREAD_SCOPE_SYSTEM scheduling contention scope, or both.

Scheduling Allocation Domain

Implementations shall support scheduling allocation domains containing one or more processors. It should be noted that the presence of multiple processors does not automatically indicate a scheduling allocation domain size greater than one. Conforming implementations on multi-processors may map all or any subset of the CPUs to one or multiple scheduling allocation domains, and could define these scheduling allocation domains on a per-thread, per-process, or per-system basis, depending on the types of applications intended to be supported by the implementation. The scheduling allocation domain is independent of scheduling contention scope, as the scheduling contention scope merely defines the set of threads with which a thread contends for processor resources, while scheduling allocation domain defines the set of processors for which it contends. The semantics of how this contention is resolved among threads for processors is determined by the scheduling policies of the threads.

The choice of scheduling allocation domain size and the level of application control over scheduling allocation domains is implementation-defined. Conforming implementations may change the size of scheduling allocation domains and the binding of threads to scheduling allocation domains at any time.

For application threads with scheduling allocation domains of size equal to one, the scheduling rules defined for SCHED_FIFO and SCHED_RR shall be used; see **Scheduling Policies** (on page 494). All threads with system scheduling contention scope, regardless of the processes in which they reside, compete for the processor according to their priorities. Threads with process scheduling contention scope compete only with other threads with process scheduling contention scope within their process.

For application threads with scheduling allocation domains of size greater than one, the rules TSP defined for SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC shall be used in an implementation-defined manner. Each thread with system scheduling contention scope competes for the processors in its scheduling allocation domain in an implementation-defined manner according to its priority. Threads with process scheduling contention scope are scheduled relative to other threads within the same scheduling contention scope in the process.

TSP If _POSIX_THREAD_SPORADIC_SERVER is defined, the rules defined for SCHED_SPORADIC in **Scheduling Policies** (on page 494) shall be used in an implementation-defined manner for application threads whose scheduling allocation domain size is greater than one.

Scheduling Documentation

If _POSIX_PRIORITY_SCHEDULING is defined, then any scheduling policies beyond TSP SCHED_OTHER, SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC, as well as the effects of the scheduling policies indicated by these other values, and the attributes required in order to support such a policy, are implementation-defined. Furthermore, the implementation shall document the effect of all processor scheduling allocation domain values supported for these policies.

2.9.5 Thread Cancelation

The thread cancelation mechanism allows a thread to terminate the execution of any other thread in the process in a controlled manner. The target thread (that is, the one that is being canceled) is allowed to hold cancelation requests pending in a number of ways and to perform application-specific cleanup processing when the notice of cancelation is acted upon.

Cancelation is controlled by the cancelation control functions. Each thread maintains its own cancelability state. Cancelation may only occur at cancelation points or when the thread is asynchronously cancelable.

The thread cancelation mechanism described in this section depends upon programs having set *deferred cancelability* state, which is specified as the default. Applications shall also carefully follow static lexical scoping rules in their execution behavior. For example, use of *setjmp*( ), *return*, *goto*, and so on, to leave user-defined cancelation scopes without doing the necessary scope pop operation results in undefined behavior.

Use of asynchronous cancelability while holding resources which potentially need to be released may result in resource loss. Similarly, cancelation scopes may only be safely manipulated (pushed and popped) when the thread is in the *deferred* or *disabled* cancelability states.

2.9.5.1 Cancelability States

The cancelability state of a thread determines the action taken upon receipt of a cancelation request. The thread may control cancelation in a number of ways.

Each thread maintains its own cancelability state, which may be encoded in two bits:

1. Cancelability-Enable: When cancelability is PTHREAD_CANCEL_DISABLE (as defined in the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**), cancelation requests against the target thread are held pending. By default, cancelability is set to PTHREAD_CANCEL_ENABLE (as defined in **<pthread.h>**).

2. Cancelability Type: When cancelability is enabled and the cancelability type is PTHREAD_CANCEL_ASYNCHRONOUS (as defined in **<pthread.h>**), new or pending cancelation requests may be acted upon at any time. When cancelability is enabled and the cancelability type is PTHREAD_CANCEL_DEFERRED (as defined in **<pthread.h>**),

cancelation requests are held pending until a cancelation point (see below) is reached. If cancelability is disabled, the setting of the cancelability type has no immediate effect as all cancelation requests are held pending; however, once cancelability is enabled again the new type is in effect. The cancelability type is PTHREAD_CANCEL_DEFERRED in all newly created threads including the thread in which *main*( ) was first invoked.

2.9.5.2 *Cancelation Points*

Cancelation points shall occur when a thread is executing the following functions: |

accept( )	*mq_timedsend*( )	*putpmsg*( )	*sigsuspend*( )
aio_suspend( )	*msgrcv*( )	*pwrite*( )	*sigtimedwait*( )
clock_nanosleep( )	*msgsnd*( )	*read*( )	*sigwait*( )
close( )	*msync*( )	*readv*( )	*sigwaitinfo*( )
connect( )	*nanosleep*( )	*recv*( )	*sleep*( )
creat( )	*open*( )	*recvfrom*( )	*system*( )
fcntl( )[2]	*pause*( )	*recvmsg*( )	*tcdrain*( )
fsync( )	*poll*( )	*select*( )	*usleep*( )
getmsg( )	*pread*( )	*sem_timedwait*( )	*wait*( )
getpmsg( )	*pthread_cond_timedwait*( )	*sem_wait*( )	*waitid*( )
lockf( )	*pthread_cond_wait*( )	*send*( )	*waitpid*( )
mq_receive( )	*pthread_join*( )	*sendmsg*( )	*write*( )
mq_send( )	*pthread_testcancel*( )	*sendto*( )	*writev*( )
mq_timedreceive( )	*putmsg*( )	*sigpause*( )	

---

2. When the *cmd* argument is F_SETLKW.

A cancelation point may also occur when a thread is executing the following functions:

catclose( )	*ftell*( )	*getwc*( )	*pthread_rwlock_wrlock*( )
catgets( )	*ftello*( )	*getwchar*( )	*putc*( )
catopen( )	*ftw*( )	*getwd*( )	*putc_unlocked*( )
closedir( )	*fwprintf*( )	*glob*( )	*putchar*( )
closelog( )	*fwrite*( )	*iconv_close*( )	*putchar_unlocked*( )
ctermid( )	*fwscanf*( )	*iconv_open*( )	*puts*( )
dbm_close( )	*getc*( )	*ioctl*( )	*pututxline*( )
dbm_delete( )	*getc_unlocked*( )	*lseek*( )	*putwc*( )
dbm_fetch( )	*getchar*( )	*mkstemp*( )	*putwchar*( )
dbm_nextkey( )	*getchar_unlocked*( )	*nftw*( )	*readdir*( )
dbm_open( )	*getcwd*( )	*opendir*( )	*readdir_r*( )
dbm_store( )	*getdate*( )	*openlog*( )	*remove*( )
dlclose( )	*getgrent*( )	*pclose*( )	*rename*( )
dlopen( )	*getgrgid*( )	*perror*( )	*rewind*( )
endgrent( )	*getgrgid_r*( )	*popen*( )	*rewinddir*( )
endhostent( )	*getgrnam*( )	*posix_fadvise*( )	*scanf*( )
endnetent( )	*getgrnam_r*( )	*posix_fallocate*( )	*seekdir*( )
endprotoent( )	*gethostbyaddr*( )	*posix_madvise*( )	*semop*( )
endpwent( )	*gethostbyname*( )	*posix_spawn*( )	*setgrent*( )
endservent( )	*gethostent*( )	*posix_spawnp*( )	*sethostent*( )
endutxent( )	*gethostname*( )	*posix_trace_clear*( )	*setnetent*( )
fclose( )	*getlogin*( )	*posix_trace_close*( )	*setprotoent*( )
fcntl( )[3]	*getlogin_r*( )	*posix_trace_create*( )	*setpwent*( )
fflush( )	*getnetbyaddr*( )	*posix_trace_create_withlog*( )	*setservent*( )
fgetc( )	*getnetbyname*( )	*posix_trace_eventtypelist_getnext_id*( )	*setutxent*( )
fgetpos( )	*getnetent*( )	*posix_trace_eventtypelist_rewind*( )	*strerror*( )
fgets( )	*getprotobyname*( )	*posix_trace_flush*( )	*syslog*( )
fgetwc( )	*getprotobynumber*( )	*posix_trace_get_attr*( )	*tmpfile*( )
fgetws( )	*getprotoent*( )	*posix_trace_get_filter*( )	*tmpnam*( )
fopen( )	*getpwent*( )	*posix_trace_get_status*( )	*ttyname*( )
fprintf( )	*getpwnam*( )	*posix_trace_getnext_event*( )	*ttyname_r*( )
fputc( )	*getpwnam_r*( )	*posix_trace_open*( )	*ungetc*( )
fputs( )	*getpwuid*( )	*posix_trace_rewind*( )	*ungetwc*( )
fputwc( )	*getpwuid_r*( )	*posix_trace_set_filter*( )	*unlink*( )
fputws( )	*gets*( )	*posix_trace_shutdown*( )	*vfprintf*( )
fread( )	*getservbyname*( )	*posix_trace_timedgetnext_event*( )	*vfwprintf*( )
freopen( )	*getservbyport*( )	*posix_typed_mem_open*( )	*vprintf*( )
fscanf( )	*getservent*( )	*printf*( )	*vwprintf*( )
fseek( )	*getutxent*( )	*pthread_rwlock_rdlock*( )	*wprintf*( )
fseeko( )	*getutxid*( )	*pthread_rwlock_timedrdlock*( )	*wscanf*( )
fsetpos( )	*getutxline*( )	*pthread_rwlock_timedwrlock*( )	

An implementation shall not introduce cancelation points into any other functions specified in this volume of IEEE Std 1003.1-200x.

---

3. For any value of the *cmd* argument.

The side effects of acting upon a cancelation request while suspended during a call of a function are the same as the side effects that may be seen in a single-threaded program when a call to a function is interrupted by a signal and the given function returns [EINTR]. Any such side effects occur before any cancelation cleanup handlers are called.

Whenever a thread has cancelability enabled and a cancelation request has been made with that thread as the target, and the thread then calls any function that is a cancelation point (such as *pthread_testcancel*( ) or *read*( )), the cancelation request shall be acted upon before the function returns. If a thread has cancelability enabled and a cancelation request is made with the thread as a target while the thread is suspended at a cancelation point, the thread shall be awakened and the cancelation request shall be acted upon. However, if the thread is suspended at a cancelation point and the event for which it is waiting occurs before the cancelation request is acted upon, it is unspecified whether the cancelation request is acted upon or whether the cancelation request remains pending and the thread resumes normal execution.

2.9.5.3 Thread Cancelation Cleanup Handlers

Each thread maintains a list of cancelation cleanup handlers. The programmer uses the *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ) functions to place routines on and remove routines from this list.

When a cancelation request is acted upon, the routines in the list are invoked one by one in LIFO sequence; that is, the last routine pushed onto the list (Last In) is the first to be invoked (First Out). The thread invokes the cancelation cleanup handler with cancelation disabled until the last cancelation cleanup handler returns. When the cancelation cleanup handler for a scope is invoked, the storage for that scope remains valid. If the last cancelation cleanup handler returns, thread execution is terminated and a status of PTHREAD_CANCELED is made available to any threads joining with the target. The symbolic constant PTHREAD_CANCELED expands to a constant expression of type (**void ***) whose value matches no pointer to an object in memory nor the value NULL.

The cancelation cleanup handlers are also invoked when the thread calls *pthread_exit*( ).

A side effect of acting upon a cancelation request while in a condition variable wait is that the mutex is re-acquired before calling the first cancelation cleanup handler. In addition, the thread is no longer considered to be waiting for the condition and the thread shall not have consumed any pending condition signals on the condition.

A cancelation cleanup handler cannot exit via *longjmp*( ) or *siglongjmp*( ).

2.9.5.4 Async-Cancel Safety

The *pthread_cancel*( ), *pthread_setcancelstate*( ), and *pthread_setcanceltype*( ) functions are defined to be async-cancel safe.

No other functions in this volume of IEEE Std 1003.1-200x are required to be async-cancel-safe.

2.9.6 Thread Read-Write Locks

Multiple readers, single writer (read-write) locks allow many threads to have simultaneous read-only access to data while allowing only one thread to have exclusive write access at any given time. They are typically used to protect data that is read more frequently than it is changed.

One or more readers acquire read access to the resource by performing a read lock operation on the associated read-write lock. A writer acquires exclusive write access by performing a write lock operation. Basically, all readers exclude any writers and a writer excludes all readers and any other writers.

A thread that has blocked on a read-write lock (for example, has not yet returned from a *pthread_rwlock_rdlock*( ) or *pthread_rwlock_wrlock*( ) call) shall not prevent any unblocked thread that is eligible to use the same processing resources from eventually making forward progress in its execution. Eligibility for processing resources shall be determined by the scheduling policy.

Read-write locks can be used to synchronize threads in the current process and other processes if they are allocated in memory that is writable and shared among the cooperating processes and have been initialized for this behavior.

2.9.7 Thread Interactions with Regular File Operations

All of the functions *chmod*( ), *close*( ), *fchmod*( ), *fcntl*( ), *fstat*( ), *ftruncate*( ), *lseek*( ), *open*( ), *read*( ), *readlink*( ), *stat*( ), *symlink*( ), and *write*( ) shall be atomic with respect to each other in the effects specified in IEEE Std 1003.1-200x when they operate on regular files. If two threads each call one of these functions, each call shall either see all of the specified effects of the other call, or none of them.

2.10 Sockets

A socket is an endpoint for communication using the facilities described in this section. A socket is created with a specific socket type, described in Section 2.10.6 (on page 509), and is associated with a specific protocol, detailed in Section 2.10.3 (on page 509). A socket is accessed via a file descriptor obtained when the socket is created. |

2.10.1 Address Families |

All network protocols are associated with a specific address family. An address family provides basic services to the protocol implementation to allow it to function within a specific network environment. These services may include packet fragmentation and reassembly, routing, addressing, and basic transport. An address family is normally comprised of a number of protocols, one per socket type. Each protocol is characterized by an abstract socket type. It is not required that an address family support all socket types. An address family may contain multiple protocols supporting the same socket abstraction. |

Section 2.10.17 (on page 516), Section 2.10.19 (on page 517), and Section 2.10.20 (on page 517), respectively, describe the use of sockets for local UNIX connections, for Internet protocols based on IPv4, and for Internet protocols based on IPv6. |

2.10.2 Addressing

An address family defines the format of a socket address. All network addresses are described using a general structure, called a **sockaddr**, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**. However, each address family imposes finer and more specific structure, generally defining a structure with fields specific to the address family. The field *sa_family* in the **sockaddr** structure contains the address family identifier, specifying the format of the *sa_data* area. The size of the *sa_data* area is unspecified.

2.10.3 Protocols

A protocol supports one of the socket abstractions detailed in Section 2.10.6. Selecting a protocol involves specifying the address family, socket type, and protocol number to the *socket*( ) function. Certain semantics of the basic socket abstractions are protocol-specific. All protocols are expected to support the basic model for their particular socket type, but may, in addition, provide non-standard facilities or extensions to a mechanism.

2.10.4 Routing

Sockets provides packet routing facilities. A routing information database is maintained, which is used in selecting the appropriate network interface when transmitting packets.

2.10.5 Interfaces

Each network interface in a system corresponds to a path through which messages can be sent and received. A network interface usually has a hardware device associated with it, though certain interfaces such as the loopback interface, do not.

2.10.6 Socket Types

A socket is created with a specific type, which defines the communication semantics and which RS allows the selection of an appropriate communication protocol. Four types are defined: SOCK_RAW, SOCK_STREAM, SOCK_SEQPACKET, and SOCK_DGRAM. Implementations may specify additional socket types.

The SOCK_STREAM socket type provides reliable, sequenced, full-duplex octet streams between the socket and a peer to which the socket is connected. A socket of type SOCK_STREAM must be in a connected state before any data may be sent or received. Record boundaries are not maintained; data sent on a stream socket using output operations of one size may be received using input operations of smaller or larger sizes without loss of data. Data may be buffered; successful return from an output function does not imply that the data has been delivered to the peer or even transmitted from the local system. If data cannot be successfully transmitted within a given time then the connection is considered broken, and subsequent operations shall fail. A SIGPIPE signal is raised if a thread sends on a broken stream (one that is no longer connected). Support for an out-of-band data transmission facility is protocol-specific.

The SOCK_SEQPACKET socket type is similar to the SOCK_STREAM type, and is also connection-oriented. The only difference between these types is that record boundaries are maintained using the SOCK_SEQPACKET type. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers parts of more than one record. Record boundaries are visible to the receiver via the MSG_EOR flag in the received message flags returned by the *recvmsg*( ) function. It is protocol-specific whether a maximum record size is imposed.

The SOCK_DGRAM socket type supports connectionless data transfer which is not necessarily acknowledged or reliable. Datagrams may be sent to the address specified (possibly multicast or

broadcast) in each output operation, and incoming datagrams may be received from multiple sources. The source address of each datagram is available when receiving the datagram. An application may also pre-specify a peer address, in which case calls to output functions shall send to the pre-specified peer. If a peer has been specified, only datagrams from that peer shall be received. A datagram must be sent in a single output operation, and must be received in a single input operation. The maximum size of a datagram is protocol-specific; with some protocols, the limit is implementation-defined. Output datagrams may be buffered within the system; thus, a successful return from an output function does not guarantee that a datagram is actually sent or received. However, implementations should attempt to detect any errors possible before the return of an output function, reporting any error by an unsuccessful return value.

RS The SOCK_RAW socket type is similar to the SOCK_DGRAM type. It differs in that it is normally used with communication providers that underlie those used for the other socket types. For this reason, the creation of a socket with type SOCK_RAW shall require appropriate privilege. The format of datagrams sent and received with this socket type generally include specific protocol headers, and the formats are protocol-specific and implementation-defined.

2.10.7 Socket I/O Mode

The I/O mode of a socket is described by the O_NONBLOCK file status flag which pertains to the open file description for the socket. This flag is initially off when a socket is created, but may be set and cleared by the use of the F_SETFL command of the *fcntl*( ) function.

When the O_NONBLOCK flag is set, functions that would normally block until they are complete shall either return immediately with an error, or shall complete asynchronously to the execution of the calling process. Data transfer operations (the *read*( ), *write*( ), *send*( ), and *recv*( ) functions) shall complete immediately, transfer only as much as is available, and then return without blocking, or return an error indicating that no transfer could be made without blocking. The *connect*( ) function initiates a connection and shall return without blocking when O_NONBLOCK is set; it shall return the error [EINPROGRESS] to indicate that the connection was initiated successfully, but that it has not yet completed.

2.10.8 Socket Owner

The owner of a socket is unset when a socket is created. The owner may be set to a process ID or process group ID using the F_SETOWN command of the *fcntl*( ) function.

2.10.9 Socket Queue Limits

The transmit and receive queue sizes for a socket are set when the socket is created. The default sizes used are both protocol-specific and implementation-defined. The sizes may be changed using the *setsockopt*( ) function.

2.10.10 Pending Error

Errors may occur asynchronously, and be reported to the socket in response to input from the network protocol. The socket stores the pending error to be reported to a user of the socket at the next opportunity. The error is returned in response to a subsequent *send*( ), *recv*( ), or *getsockopt*( ) operation on the socket, and the pending error is then cleared.

2.10.11 Socket Receive Queue

A socket has a receive queue that buffers data when it is received by the system until it is removed by a receive call. Depending on the type of the socket and the communication provider, the receive queue may also contain ancillary data such as the addressing and other protocol data associated with the normal data in the queue, and may contain out-of-band or expedited data. The limit on the queue size includes any normal, out-of-band data, datagram source addresses, and ancillary data in the queue. The description in this section applies to all sockets, even though some elements cannot be present in some instances.

The contents of a receive buffer are logically structured as a series of data segments with associated ancillary data and other information. A data segment may contain normal data or out-of-band data, but never both. A data segment may complete a record if the protocol supports records (always true for types SOCK_SEQPACKET and SOCK_DGRAM). A record may be stored as more than one segment; the complete record might never be present in the receive buffer at one time, as a portion might already have been returned to the application, and another portion might not yet have been received from the communications provider. A data segment may contain ancillary protocol data, which is logically associated with the segment. Ancillary data is received as if it were queued along with the first normal data octet in the segment (if any). A segment may contain ancillary data only, with no normal or out-of-band data. For the purposes of this section, a datagram is considered to be a data segment that terminates a record, and that includes a source address as a special type of ancillary data. Data segments are placed into the queue as data is delivered to the socket by the protocol. Normal data segments are placed at the end of the queue as they are delivered. If a new segment contains the same type of data as the preceding segment and includes no ancillary data, and if the preceding segment does not terminate a record, the segments are logically merged into a single segment.

The receive queue is logically terminated if an end-of-file indication has been received or a connection has been terminated. A segment shall be considered to be terminated if another segment follows it in the queue, if the segment completes a record, or if an end-of-file or other connection termination has been reported. The last segment in the receive queue shall also be considered to be terminated while the socket has a pending error to be reported.

A receive operation shall never return data or ancillary data from more than one segment.

2.10.12 Socket Out-of-Band Data State

The handling of received out-of-band data is protocol-specific. Out-of-band data may be placed in the socket receive queue, either at the end of the queue or before all normal data in the queue. In this case, out-of-band data is returned to an application program by a normal receive call. Out-of-band data may also be queued separately rather than being placed in the socket receive queue, in which case it shall be returned only in response to a receive call that requests out-of-band data. It is protocol-specific whether an out-of-band data mark is placed in the receive queue to demarcate data preceding the out-of-band data and following the out-of-band data. An out-of-band data mark is logically an empty data segment that cannot be merged with other segments in the queue. An out-of-band data mark is never returned in response to an input operation. The *sockatmark*( ) function can be used to test whether an out-of-band data mark is the first element in the queue. If an out-of-band data mark is the first element in the queue when an input function is called without the MSG_PEEK option, the mark is removed from the queue and the following data (if any) is processed as if the mark had not been present.

2.10.13 Connection Indication Queue

Sockets that are used to accept incoming connections maintain a queue of outstanding connection indications. This queue is a list of connections that are awaiting acceptance by the application; see *listen*( ).

2.10.14 Signals

One category of event at the socket interface is the generation of signals. These signals report protocol events or process errors relating to the state of the socket. The generation or delivery of a signal does not change the state of the socket, although the generation of the signal may have been caused by a state change.

The SIGPIPE signal shall be sent to a thread that attempts to send data on a socket that is no longer able to send. In addition, the send operation fails with the error [EPIPE].

If a socket has an owner, the SIGURG signal is sent to the owner of the socket when it is notified of expedited or out-of-band data. The socket state at this time is protocol-dependent, and the status of the socket is specified in Section 2.10.17 (on page 516), Section 2.10.19 (on page 517), and Section 2.10.20 (on page 517). Depending on the protocol, the expedited data may or may not have arrived at the time of signal generation.

2.10.15 Asynchronous Errors

If any of the following conditions occur asynchronously for a socket, the corresponding value listed below shall become the pending error for the socket:

[ECONNABORTED]
The connection was aborted locally.

[ECONNREFUSED]
For a connection-mode socket attempting a non-blocking connection, the attempt to connect was forcefully rejected. For a connectionless-mode socket, an attempt to deliver a datagram was forcefully rejected.

[ECONNRESET]
The peer has aborted the connection.

[EHOSTDOWN]
The destination host has been determined to be down or disconnected.

[EHOSTUNREACH]
The destination host is not reachable.

[EMSGSIZE]
For a connectionless-mode socket, the size of a previously sent datagram prevented delivery.

[ENETDOWN]
The local network connection is not operational.

[ENETRESET]
The connection was aborted by the network.

[ENETUNREACH]
The destination network is not reachable.

2.10.16 Use of Options

There are a number of socket options which either specialize the behavior of a socket or provide useful information. These options may be set at different protocol levels and are always present at the uppermost ‘‘socket’’ level.

Socket options are manipulated by two functions, *getsockopt*( ) and *setsockopt*( ). These functions allow an application program to customize the behavior and characteristics of a socket to provide the desired effect.

All of the options have default values. The type and meaning of these values is defined by the protocol level to which they apply. Instead of using the default values, an application program may choose to customize one or more of the options. However, in the bulk of cases, the default values are sufficient for the application.

Some of the options are used to enable or disable certain behavior within the protocol modules (for example, turn on debugging) while others may be used to set protocol-specific information (for example, IP time-to-live on all the application’s outgoing packets). As each of the options is introduced, its effect on the underlying protocol modules is described.

Table 2-1 shows the value for the socket level.

Table 2-1 Value of Level for Socket Options

Name	Description
SOL_SOCKET	Options are intended for the sockets level.

Table 2-2 (on page 514) lists those options present at the socket level; that is, when the *level* parameter of the *getsockopt*( ) or *setsockopt*( ) function is SOL_SOCKET, the types of the option value parameters associated with each option, and a brief synopsis of the meaning of the option value parameter. Unless otherwise noted, each may be examined with *getsockopt*( ) and set with *setsockopt*( ) on all types of socket.

Table 2-2 Socket-Level Options

Option	Parameter Type	Parameter Meaning
SO_BROADCAST	**int**	Non-zero requests permission to transmit broadcast datagrams (SOCK_DGRAM sockets only).
SO_DEBUG	**int**	Non-zero requests debugging in underlying protocol modules.
SO_DONTROUTE	**int**	Non-zero requests bypass of normal routing; route based on destination address only.
SO_ERROR	**int**	Requests and clears pending error information on the socket (*getsockopt*( ) only).
SO_KEEPALIVE	**int**	Non-zero requests periodic transmission of keepalive messages (protocol-specific).
SO_LINGER	**struct linger**	Specify actions to be taken for queued, unsent data on *close*( ): linger on/off and linger time in seconds.
SO_OOBINLINE	**int**	Non-zero requests that out-of-band data be placed into normal data input queue as received.
SO_RCVBUF	**int**	Size of receive buffer (in bytes).
SO_RCVLOWAT	**int**	Minimum amount of data to return to application for input operations (in bytes).
SO_RCVTIMEO	**struct timeval**	Timeout value for a socket receive operation.
SO_REUSEADDR	**int**	Non-zero requests reuse of local addresses in *bind*( ) (protocol-specific).
SO_SNDBUF	**int**	Size of send buffer (in bytes).
SO_SNDLOWAT	**int**	Minimum amount of data to send for output operations (in bytes).
SO_SNDTIMEO	**struct timeval**	Timeout value for a socket send operation.
SO_TYPE	**int**	Identify socket type (*getsockopt*( ) only).

The SO_BROADCAST option requests permission to send broadcast datagrams on the socket. Support for SO_BROADCAST is protocol-specific. The default for SO_BROADCAST is that the ability to send broadcast datagrams on a socket is disabled.

The SO_DEBUG option enables debugging in the underlying protocol modules. This can be useful for tracing the behavior of the underlying protocol modules during normal system operation. The semantics of the debug reports are implementation-defined. The default value for SO_DEBUG is for debugging to be turned off.

The SO_DONTROUTE option requests that outgoing messages bypass the standard routing facilities. The destination must be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. It is protocol-specific whether this option has any effect and how the outgoing network interface is chosen. Support for this option with each protocol is implementation-defined.

The SO_ERROR option is used only on *getsockopt*( ). When this option is specified, *getsockopt*( ) shall return any pending error on the socket and clear the error status. It shall return a value of 0 if there is no pending error. SO_ERROR may be used to check for asynchronous errors on connected connectionless-mode sockets or for other types of asynchronous errors. SO_ERROR has no default value.

The SO_KEEPALIVE option enables the periodic transmission of messages on a connected socket. The behavior of this option is protocol-specific. The default value for SO_KEEPALIVE is zero, specifying that this capability is turned off.

The SO_LINGER option controls the action of the interface when unsent messages are queued on a socket and a *close*( ) is performed. The details of this option are protocol-specific. The default value for SO_LINGER is zero, or off, for the *l_onoff* element of the option value and zero seconds for the linger time specified by the *l_linger* element.

The SO_OOBINLINE option is valid only on protocols that support out-of-band data. The SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it is then accessible using the *read*( ) or *recv*( ) functions without the MSG_OOB flag set. The default for SO_OOBINLINE is off; that is, for out-of-band data not to be placed in the normal data input queue.

The SO_RCVBUF option requests that the buffer space allocated for receive operations on this socket be set to the value, in bytes, of the option value. Applications may wish to increase buffer size for high volume connections, or may decrease buffer size to limit the possible backlog of incoming data. The default value for the SO_RCVBUF option value is implementation-defined, and may vary by protocol.

The maximum value for the option for a socket may be obtained by the use of the *fpathconf*( ) function, using the value _PC_SOCK_MAXBUF.

The SO_RCVLOWAT option sets the minimum number of bytes to process for socket input operations. In general, receive calls block until any (non-zero) amount of data is received, then return the smaller of the amount available or the amount requested. The default value for SO_RCVLOWAT is 1, and does not affect the general case. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that returned (for example, out-of-band data). As mentioned previously, the default value for SO_RCVLOWAT is 1 byte. It is implementation-defined whether the SO_RCVLOWAT option can be set.

The SO_RCVTIMEO option is an option to set a timeout value for input operations. It accepts a **timeval** structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it shall return with a partial count or *errno* shall be set to [EWOULDBLOCK] if no data were received. The default for this option is the value zero, which indicates that a receive operation will not timeout. It is implementation-defined whether the SO_RCVTIMEO option can be set.

The SO_REUSEADDR option indicates that the rules used in validating addresses supplied in a *bind*( ) should allow reuse of local addresses. Operation of this option is protocol-specific. The default value for SO_REUSEADDR is off; that is, reuse of local addresses is not permitted.

The SO_SNDBUF option requests that the buffer space allocated for send operations on this socket be set to the value, in bytes, of the option value. The default value for the SO_SNDBUF option value is implementation-defined, and may vary by protocol. The maximum value for the option for a socket may be obtained by the use of the *fpathconf*( ) function, using the value _PC_SOCK_MAXBUF.

The SO_SNDLOWAT option sets the minimum number of bytes to process for socket output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Non-blocking output operations process as much data as permitted subject to flow control without blocking, but

process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. A *select*( ) operation testing the ability to write to a socket shall return true only if the send low water mark could be processed. The default value for SO_SNDLOWAT is implementation-defined and protocol-specific. It is implementation-defined whether the SO_SNDLOWAT option can be set.

The SO_SNDTIMEO option is an option to set a timeout value for the amount of time that an output function shall block because flow control prevents data from being sent. As noted in Table 2-2 (on page 514), the option value is a **timeval** structure with the number of seconds and microseconds specifying the limit on how long to wait for an output operation to complete. If a send operation has blocked for this much time, it shall return with a partial count or *errno* set to [EWOULDBLOCK] if no data were sent. The default for this option is the value zero, which indicates that a send operation will not timeout. It is implementation-defined whether the SO_SNDTIMEO option can be set.

The SO_TYPE option is used only on *getsockopt*( ). When this option is specified, *getsockopt*( ) shall return the type of the socket (for example, SOCK_STREAM). This option is useful to servers that inherit sockets on start-up. SO_TYPE has no default value.

2.10.17 Use of Sockets for Local UNIX Connections

Support for UNIX domain sockets is mandatory.

UNIX domain sockets provide process-to-process communication in a single system.

2.10.17.1 Headers

The symbolic constant AF_UNIX defined in the **<sys/socket.h>** header is used to identify the UNIX domain address family. The **<sys/un.h>** header contains other definitions used in connection with UNIX domain sockets. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers.

The **sockaddr_storage** structure defined in **<sys/socket.h>** shall be large enough to accommodate a **sockaddr_un** structure (see the **<sys/un.h>** header defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers) and shall be aligned at an appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_un** structures and used to access the fields of those structures without alignment problems. When a **sockaddr_storage** structure is cast as a **sockaddr_un** structure, the *ss_family* field maps onto the *sun_family* field.

2.10.18 Use of Sockets over Internet Protocols

When a socket is created in the Internet family with a protocol value of zero, the implementation shall use the protocol listed below for the type of socket created.

SOCK_STREAM	IPPROTO_TCP.
SOCK_DGRAM	IPPROTO_UDP.
RS SOCK_RAW	IPPROTO_RAW.
SOCK_SEQPACKET	Unspecified.

RS A raw interface to IP is available by creating an Internet socket of type SOCK_RAW. The default protocol for type SOCK_RAW shall be identified in the IP header with the value IPPROTO_RAW. Applications should not use the default protocol when creating a socket with type SOCK_RAW, but should identify a specific protocol by value. The ICMP control protocol is accessible from a raw socket by specifying a value of IPPROTO_ICMP for protocol.

2.10.19 Use of Sockets over Internet Protocols Based on IPv4

Support for sockets over Internet protocols based on IPv4 is mandatory.

2.10.19.1 Headers

The symbolic constant AF_INET defined in the **<sys/socket.h>** header is used to identify the IPv4 Internet address family. The **<netinet/in.h>** header contains other definitions used in connection with IPv4 Internet sockets. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers.

The **sockaddr_storage** structure defined in **<sys/socket.h>** shall be large enough to accommodate a **sockaddr_in** structure (see the **<netinet/in.h>** header defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers) and shall be aligned at an appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_in** structures and used to access the fields of those structures without alignment problems. When a **sockaddr_storage** structure is cast as a **sockaddr_in** structure, the *ss_family* field maps onto the *sin_family* field.

2.10.20 Use of Sockets over Internet Protocols Based on IPv6

IP6 This section describes extensions to support sockets over Internet protocols based on IPv6. This functionality is dependent on support of the IPV6 option (and the rest of this section is not further shaded for this option).

To enable smooth transition from IPv4 to IPv6, the features defined in this section may, in certain circumstances, also be used in connection with IPv4; see Section 2.10.20.2 (on page 518).

2.10.20.1 Addressing

IPv6 overcomes the addressing limitations of previous versions by using 128-bit addresses instead of 32-bit addresses. The IPv6 address architecture is described in RFC 2373.

There are three kinds of IPv6 address:

Unicast
: Identifies a single interface.

  A unicast address can be global, link-local (designed for use on a single link), or site-local (designed for systems not connected to the Internet). Link-local and site-local addresses need not be globally unique.

Anycast
: Identifies a set of interfaces such that a packet sent to the address can be delivered to any member of the set.

  An anycast address is similar to a unicast address; the nodes to which an anycast address is assigned must be explicitly configured to know that it is an anycast address.

Multicast
: Identifies a set of interfaces such that a packet sent to the address should be delivered to every member of the set.

  An application can send multicast datagrams by simply specifying an IPv6 multicast address in the *address* argument of *sendto*( ). To receive multicast datagrams, an application must join the multicast group (using *setsockopt*( ) with IPV6_JOIN_GROUP) and must bind to the socket the UDP port on which datagrams will be received. Some applications should also bind the multicast group address to the socket, to prevent other datagrams destined to that port from being delivered to the socket.

A multicast address can be global, node-local, link-local, site-local, or organization-local.

The following special IPv6 addresses are defined:

Unspecified
: An address that is not assigned to any interface and is used to indicate the absence of an address.

Loopback
: A unicast address that is not assigned to any interface and can be used by a node to send packets to itself.

Two sets of IPv6 addresses are defined to correspond to IPv4 addresses:

IPv4-compatible addresses
: These are assigned to nodes that support IPv6 and can be used when traffic is ‘‘tunneled’’ through IPv4.

IPv4-mapped addresses
: These are used to represent IPv4 addresses in IPv6 address format; see Section 2.10.20.2.

Note that the unspecified address and the loopback address must not be treated as IPv4-compatible addresses.

2.10.20.2 Compatibility with IPv4

The API provides the ability for IPv6 applications to interoperate with applications using IPv4, by using IPv4-mapped IPv6 addresses. These addresses can be generated automatically by the *getaddrinfo*( ) function when the specified host has only IPv4 addresses.

Applications may use AF_INET6 sockets to open TCP connections to IPv4 nodes, or send UDP packets to IPv4 nodes, by simply encoding the destination’s IPv4 address as an IPv4-mapped IPv6 address, and passing that address, within a **sockaddr_in6** structure, in the *connect*( ), *sendto*( ) or *sendmsg*( ) function. When applications use AF_INET6 sockets to accept TCP connections from IPv4 nodes, or receive UDP packets from IPv4 nodes, the system shall return the peer’s address to the application in the *accept*( ), *recvfrom*( ), *recvmsg*( ), or *getpeername*( ) function using a **sockaddr_in6** structure encoded this way. If a node has an IPv4 address, then the implementation may allow applications to communicate using that address via an AF_INET6 socket. In such a case, the address will be represented at the API by the corresponding IPv4-mapped IPv6 address. Also, the implementation may allow an AF_INET6 socket bound to **in6addr_any** to receive inbound connections and packets destined to one of the node’s IPv4 addresses.

An application may use AF_INET6 sockets to bind to a node’s IPv4 address by specifying the address as an IPv4-mapped IPv6 address in a **sockaddr_in6** structure in the *bind*( ) function. For an AF_INET6 socket bound to a node’s IPv4 address, the system shall return the address in the *getsockname*( ) function as an IPv4-mapped IPv6 address in a **sockaddr_in6** structure.

2.10.20.3 Interface Identification

Each local interface is assigned a unique positive integer as a numeric index. Indexes start at 1; zero is not used. There may be gaps so that there is no current interface for a particular positive index. Each interface also has a unique implementation-defined name.

2802 *2.10.20.4 Options*

2803 The following options apply at the IPPROTO_IPV6 level:

2804 IPV6_JOIN_GROUP
2805 When set via *setsockopt*( ), it joins the application to a multicast group on an interface
2806 (identified by its index) and addressed by a given multicast address, enabling packets sent
2807 to that address to be read via the socket. If the interface index is specified as zero, the
2808 system selects the interface (for example, by looking up the address in a routing table and
2809 using the resulting interface).

2810 An attempt to read this option using *getsockopt*( ) shall result in an [EOPNOTSUPP] error.

2811 The value of this option is an **ipv6_mreq** structure.

2812 IPV6_LEAVE_GROUP
2813 When set via *setsockopt*( ), it removes the application from the multicast group on an
2814 interface (identified by its index) and addressed by a given multicast address.

2815 An attempt to read this option using *getsockopt*( ) shall result in an [EOPNOTSUPP] error.

2816 The value of this option is an **ipv6_mreq** structure.

2817 IPV6_MULTICAST_HOPS
2818 The value of this option is the hop limit for outgoing multicast IPv6 packets sent via the
2819 socket. Its possible values are the same as those of IPV6_UNICAST_HOPS. If the
2820 IPV6_MULTICAST_HOPS option is not set, a value of 1 is assumed. This option can be set
2821 via *setsockopt*( ) and read via *getsockopt*( ).

2822 IPV6_MULTICAST_IF
2823 The index of the interface to be used for outgoing multicast packets. It can be set via
2824 *setsockopt*( ) and read via *getsockopt*( ).

2825 IPV6_MULTICAST_LOOP
2826 This option controls whether outgoing multicast packets should be delivered back to the
2827 local application when the sending interface is itself a member of the destination multicast
2828 group. If it is set to 1 they are delivered. If it is set to 0 they are not. Other values result in an
2829 [EINVAL] error. This option can be set via *setsockopt*( ) and read via *getsockopt*( ).

2830 IPV6_UNICAST_HOPS
2831 The value of this option is the hop limit for outgoing unicast IPv6 packets sent via the
2832 socket. If the option is not set, or is set to –1, the system selects a default value. Attempts to
2833 set a value less than –1 or greater than 255 shall result in an [EINVAL] error. This option can
2834 be set via *setsockopt*( ) and read via *getsockopt*( ). |

2835 IPV6_V6ONLY |
2836 This socket option restricts AF_INET6 sockets to IPv6 communications only. AF_INET6 |
2837 sockets may be used for both IPv4 and IPv6 communications. Some applications may want |
2838 to restrict their use of an AF_INET6 socket to IPv6 communications only. For these |
2839 applications, the IPv6_V6ONLY socket option is defined. When this option is turned on, the |
2840 socket can be used to send and receive IPv6 packets only. This is an IPPROTO_IPV6-level |
2841 option. This option takes an **int** value. This is a Boolean option. By default, this option is |
2842 turned off. |

2843 An [EOPNOTSUPP] error shall result if IPV6_JOIN_GROUP or IPV6_LEAVE_GROUP is used
2844 with *getsockopt*( ).

2845 *2.10.20.5 Headers*

2846 The symbolic constant AF_INET6 is defined in the **<sys/socket.h>** header to identify the IPv6 |
2847 Internet address family. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13,
2848 Headers.

2849 The **sockaddr_storage** structure defined in **<sys/socket.h>** shall be large enough to
2850 accommodate a **sockaddr_in6** structure (see the **<netinet/in.h>** header defined in the Base
2851 Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers) and shall be aligned at an
2852 appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_in6** structures
2853 and used to access the fields of those structures without alignment problems. When a
2854 **sockaddr_storage** structure is cast as a **sockaddr_in6** structure, the *ss_family* field maps onto the
2855 *sin6_family* field.

2856 The **<netinet/in.h>**, **<arpa/inet.h>**, and **<netdb.h>** headers contain other definitions used in
2857 connection with IPv6 Internet sockets; see the Base Definitions volume of IEEE Std 1003.1-200x,
2858 Chapter 13, Headers.

2859 2.11 Tracing

2860 TRC This section describes extensions to support tracing of user applications. This functionality is
2861 dependent on support of the Trace option (and the rest of this section is not further shaded for
2862 this option).

2863 The tracing facilities defined in IEEE Std 1003.1-200x allow a process to select a set of trace event
2864 types, to activate a trace stream of the selected trace events as they occur in the flow of
2865 execution, and to retrieve the recorded trace events.

2866 The tracing operation relies on three logically different components: the traced process, the
2867 controller process, and the analyzer process. During the execution of the traced process, when a
2868 trace point is reached, a trace event is recorded into the trace streams created for that process in
2869 which the associated trace event type identifier is not being filtered out. The controller process
2870 controls the operation of recording the trace events into the trace stream. It shall be able to:

2871 • Initialize the attributes of a trace stream

2872 • Create the trace stream (for a specified traced process) using those attributes

2873 • Start and stop tracing for the trace stream

2874 • Filter the type of trace events to be recorded, if the Trace Event Filter option is supported

2875 • Shut a trace stream down

2876 These operations can be done for an active trace stream. The analyzer process retrieves the
2877 traced events either at runtime, when the trace stream has not yet been shut down, but is still
2878 recording trace events; or after opening a trace log that had been previously recorded and shut
2879 down. These three logically different operations can be performed by the same process, or can be
2880 distributed into different processes.

2881 A trace stream identifier can be created by a call to *posix_trace_create*(), *posix_trace_create_withlog*(), or *posix_trace_open*(). The *posix_trace_create*() and *posix_trace_create_withlog*() functions should be used by a controller process. The *posix_trace_open*() should be used by an analyzer process.

2885 The tracing functions can serve different purposes. One purpose is debugging the possibly pre-
2886 instrumented code, while another is post-mortem fault analysis. These two potential uses differ
2887 in that the first requires pre-filtering capabilities to avoid overwhelming the trace stream and

permits focusing on expected information; while the second needs comprehensive trace capabilities in order to be able to record all types of information.

The events to be traced belong to two classes:

1. User trace events (generated by the application instrumentation)
2. System trace events (generated by the operating system)

The trace interface defines several system trace event types associated with control of and operation of the trace stream. This small set of system trace events includes the minimum required to interpret correctly the trace event information present in the stream. Other desirable system trace events for some particular application profile may be implemented and are encouraged; for example, process and thread scheduling, signal occurrence, and so on.

Each traced process shall have a mapping of the trace event names to trace event type identifiers that have been defined for that process. Each active trace stream shall have a mapping that incorporates all the trace event type identifiers predefined by the trace system plus all the mappings of trace event names to trace event type identifiers of the processes that are being traced into that trace stream. These mappings are defined from the instrumented application by calling the *posix_trace_eventid_open*( ) function and from the controller process by calling the *posix_trace_trid_eventid_open*( ) function. For a pre-recorded trace stream, the list of trace event types is obtained from the pre-recorded trace log.

The *st_ctime* and *st_mtime* fields of a file associated with an active trace stream shall be marked for update every time any of the tracing operations modifies that file.

The *st_atime* field of a file associated with a trace stream shall be marked for update every time any of the tracing operations causes data to be read from that file.

Results are undefined if the application performs any operation on a file descriptor associated with an active or pre-recorded trace stream until *posix_trace_shutdown*( ) or *posix_trace_close*( ) is called for that trace stream.

The main purpose of this option is to define a complete set of functions and concepts that allow a conforming application to be traced from creation to termination, whatever its realtime constraints and properties.

2.11.1 Tracing Data Definitions

2.11.1.1 Structures

The **<trace.h>** header shall define the *posix_trace_status_info* and *posix_trace_event_info* structures described below. Implementations may add extensions to these structures.

posix_trace_status_info Structure

To facilitate control of a trace stream, information about the current state of an active trace stream can be obtained dynamically. This structure is returned by a call to the *posix_trace_get_status*( ) function.

The **posix_trace_status_info** structure defined in **<trace.h>** shall contain at least the following members:

Member Type	Member Name	Description
int	*posix_stream_status*	The operating mode of the trace stream.
int	*posix_stream_full_status*	The full status of the trace stream.
int	*posix_stream_overrun_status*	Indicates whether trace events were lost in the trace stream.

If the Trace Log option is supported in addition to the Trace option, the **posix_trace_status_info** structure defined in **<trace.h>** shall contain at least the following additional members:

Member Type	Member Name	Description
int	*posix_stream_flush_status*	Indicates whether a flush is in progress.
int	*posix_stream_flush_error*	Indicates whether any error occurred during the last flush operation.
int	*posix_log_overrun_status*	Indicates whether trace events were lost in the trace log.
int	*posix_log_full_status*	The full status of the trace log.

The *posix_stream_status* member indicates the operating mode of the trace stream and shall have one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_RUNNING
: Tracing is in progress; that is, the trace stream is accepting trace events.

POSIX_TRACE_SUSPENDED
: The trace stream is not accepting trace events. The tracing operation has not yet started or has stopped, either following a *posix_trace_stop*( ) function call or because the trace resources are exhausted.

The *posix_stream_full_status* member indicates the full status of the trace stream, and it shall have one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_FULL
: The space in the trace stream for trace events is exhausted.

POSIX_TRACE_NOT_FULL
: There is still space available in the trace stream.

The combination of the *posix_stream_status* and *posix_stream_full_status* members also indicates the actual status of the stream. The status shall be interpreted as follows:

POSIX_TRACE_RUNNING and POSIX_TRACE_NOT_FULL
: This status combination indicates that tracing is in progress, and there is space available for recording more trace events.

POSIX_TRACE_RUNNING and POSIX_TRACE_FULL
: This status combination indicates that tracing is in progress and that the trace stream is full of trace events. This status combination cannot occur unless the *stream-full-policy* is set to POSIX_TRACE_LOOP. The trace stream contains trace events recorded during a moving time window of prior trace events, and some older trace events may have been overwritten and thus lost.

POSIX_TRACE_SUSPENDED and POSIX_TRACE_NOT_FULL
: This status combination indicates that tracing has not yet been started, has been stopped by the *posix_trace_stop*( ) function, or has been cleared by the *posix_trace_clear*( ) function.

POSIX_TRACE_SUSPENDED and POSIX_TRACE_FULL
: This status combination indicates that tracing has been stopped by the implementation because the *stream-full-policy* attribute was POSIX_TRACE_UNTIL_FULL and trace resources were exhausted, or that the trace stream was stopped by the function *posix_trace_stop*( ) at a time when trace resources were exhausted.

The *posix_stream_overrun_status* member indicates whether trace events were lost in the trace stream, and shall have one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_OVERRUN
: At least one trace event was lost and thus was not recorded in the trace stream.

POSIX_TRACE_NO_OVERRUN
: No trace events were lost.

When the corresponding trace stream is created, the *posix_stream_overrun_status* member shall be set to POSIX_TRACE_NO_OVERRUN.

Whenever an overrun occurs, *posix_stream_overrun_status* member shall be set to POSIX_TRACE_OVERRUN.

An overrun occurs when:

- The policy is POSIX_TRACE_LOOP and a recorded trace event is overwritten.
- The policy is POSIX_TRACE_UNTIL_FULL and the trace stream is full when a trace event is generated.
- If the Trace Log option is supported, the policy is POSIX_TRACE_FLUSH and at least one trace event is lost while flushing the trace stream to the trace log.

The *posix_stream_overrun_status* member is reset to zero after its value is read.

If the Trace Log option is supported in addition to the Trace option, the *posix_stream_flush_status*, *posix_stream_flush_error*, *posix_log_overrun_status*, and *posix_log_full_status* members are defined as follows; otherwise, they are undefined.

The *posix_stream_flush_status* member indicates whether a flush operation is being performed and shall have one of the following values defined by manifest constants in the header **<trace.h>**:

POSIX_TRACE_FLUSHING
: The trace stream is currently being flushed to the trace log.

POSIX_TRACE_NOT_FLUSHING
: No flush operation is in progress.

The *posix_stream_flush_status* member shall be set to POSIX_TRACE_FLUSHING if a flush operation is in progress either due to a call to the *posix_trace_flush*( ) function (explicit or caused by a trace stream shutdown operation) or because the trace stream has become full with the *stream-full-policy* attribute set to POSIX_TRACE_FLUSH. The *posix_stream_flush_status* member shall be set to POSIX_TRACE_NOT_FLUSHING if no flush operation is in progress.

The *posix_stream_flush_error* member shall be set to zero if no error occurred during flushing. If an error occurred during a previous flushing operation, the *posix_stream_flush_error* member shall be set to the value of the first error that occurred. If more than one error occurs while flushing, error values after the first shall be discarded. The *posix_stream_flush_error* member is reset to zero after its value is read.

The *posix_log_overrun_status* member indicates whether trace events were lost in the trace log, and shall have one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_OVERRUN
: At least one trace event was lost.

POSIX_TRACE_NO_OVERRUN
: No trace events were lost.

When the corresponding trace stream is created, the *posix_log_overrun_status* member shall be set to POSIX_TRACE_NO_OVERRUN. Whenever an overrun occurs, this status shall be set to POSIX_TRACE_OVERRUN. The *posix_log_overrun_status* member is reset to zero after its value is read.

The *posix_log_full_status* member indicates the full status of the trace log, and it shall have one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_FULL
: The space in the trace log is exhausted.

POSIX_TRACE_NOT_FULL
: There is still space available in the trace log.

The *posix_log_full_status* member is only meaningful if the *log-full-policy* attribute is either POSIX_TRACE_UNTIL_FULL or POSIX_TRACE_LOOP.

For an active trace stream without log, that is created by the *posix_trace_create*( ) function, the *posix_log_overrun_status* member shall be set to POSIX_TRACE_NO_OVERRUN and the *posix_log_full_status* member shall be set to POSIX_TRACE_NOT_FULL.

posix_trace_event_info Structure

The trace event structure **posix_trace_event_info** contains the information for one recorded trace event. This structure is returned by the set of functions *posix_trace_getnext_event*( ), *posix_trace_timedgetnext_event*( ), and *posix_trace_trygetnext_event*( ).

The **posix_trace_event_info** structure defined in **<trace.h>** shall contain at least the following members:

Member Type	Member Name	Description
trace_event_id_t	*posix_event_id*	Trace event type identification.
pid_t	*posix_pid*	Process ID of the process that generated the trace event.
void *	*posix_prog_address*	Address at which the trace point was invoked.
int	*posix_truncation_status*	Status about the truncation of the data associated with this trace event.
struct timespec	*posix_timestamp*	Time at which the trace event was generated.

In addition, if the Trace option and the Threads option are both supported, the **posix_trace_event_info** structure defined in **<trace.h>** shall contain the following additional member:

Member Type	Member Name	Description
pthread_t	*posix_thread_id*	Thread ID of the thread that generated the trace event.

The *posix_event_id* member represents the identification of the trace event type and its value is not directly defined by the user. This identification is returned by a call to one of the following functions: *posix_trace_trid_eventid_open*( ), *posix_trace_eventtypelist_getnext_id*( ), or *posix_trace_eventid_open*( ). The name of the trace event type can be obtained by calling *posix_trace_eventid_get_name*( ).

The *posix_pid* is the process identifier of the traced process which generated the trace event. If the *posix_event_id* member is one of the implementation-defined system trace events and that trace event is not associated with any process, the *posix_pid* member shall be set to zero.

For a user trace event, the *posix_prog_address* member is the process mapped address of the point at which the associated call to the *posix_trace_event*( ) function was made. For a system trace event, if the trace event is caused by a system service explicitly called by the application, the *posix_prog_address* member shall be the address of the process at the point where the call to that system service was made.

The *posix_truncation_status* member defines whether the data associated with a trace event has been truncated at the time the trace event was generated, or at the time the trace event was read from the trace stream, or (if the Trace Log option is supported) from the trace log (see the *event* argument from the *posix_trace_getnext_event*( ) function). The *posix_truncation_status* member shall have one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_NOT_TRUNCATED
: All the traced data is available.

POSIX_TRACE_TRUNCATED_RECORD
: Data was truncated at the time the trace event was generated.

POSIX_TRACE_TRUNCATED_READ
: Data was truncated at the time the trace event was read from a trace stream or a trace log because the reader's buffer was too small. This truncation status overrides the POSIX_TRACE_TRUNCATED_RECORD status.

The *posix_timestamp* member shall be the time at which the trace event was generated. The clock used is implementation-defined, but the resolution of this clock can be retrieved by a call to the *posix_trace_attr_getclockres*( ) function.

If the Threads option is supported in addition to the Trace option:

- The *posix_thread_id* member is the identifier of the thread that generated the trace event. If the *posix_event_id* member is one of the implementation-defined system trace events and that trace event is not associated with any thread, the *posix_thread_id* member shall be set to zero.

Otherwise, this member is undefined.

2.11.1.2 Trace Stream Attributes

Trace streams have attributes that compose the **posix_trace_attr_t** trace stream attributes object. This object shall contain at least the following attributes:

- The *generation-version* attribute identifies the origin and version of the trace system.

- The *trace-name* attribute is a character string defined by the trace controller, and that identifies the trace stream.
- The *creation-time* attribute represents the time of the creation of the trace stream.
- The *clock-resolution* attribute defines the clock resolution of the clock used to generate timestamps.
- The *stream-min-size* attribute defines the minimum size in bytes of the trace stream strictly reserved for the trace events.
- The *stream-full-policy* attribute defines the policy followed when the trace stream is full; its value is POSIX_TRACE_LOOP, POSIX_TRACE_UNTIL_FULL, or POSIX_TRACE_FLUSH.
- The *max-data-size* attribute defines the maximum record size in bytes of a trace event.

In addition, if the Trace option and the Trace Inherit option are both supported, the **posix_trace_attr_t** trace stream creation attributes object shall contain at least the following attributes:

- The *inheritance* attribute specifies whether a newly created trace stream will inherit tracing in its parent's process trace stream. It is either POSIX_TRACE_INHERITED or POSIX_TRACE_CLOSE_FOR_CHILD.

In addition, if the Trace option and the Trace Log option are both supported, the **posix_trace_attr_t** trace stream creation attributes object shall contain at least the following attribute:

- If the file type corresponding to the trace log supports the POSIX_TRACE_LOOP or the POSIX_TRACE_UNTIL_FULL policies, the *log-max-size* attribute defines the maximum size in bytes of the trace log associated with an active trace stream. Other stream data—for example, trace attribute values—shall not be included in this size.
- The *log-full-policy* attribute defines the policy of a trace log associated with an active trace stream to be POSIX_TRACE_LOOP, POSIX_TRACE_UNTIL_FULL, or POSIX_TRACE_APPEND.

2.11.2 Trace Event Type Definitions

2.11.2.1 System Trace Event Type Definitions

The following system trace event types, defined in the **<trace.h>** header, track the invocation of the trace operations:

- POSIX_TRACE_START shall be associated with a trace start operation.
- POSIX_TRACE_STOP shall be associated with a trace stop operation.
- if the Trace Event Filter option is supported, POSIX_TRACE_FILTER shall be associated with a trace event type filter change operation.

The following system trace event types, defined in the **<trace.h>** header, report operational trace events:

- POSIX_TRACE_OVERFLOW shall mark the beginning of a trace overflow condition.
- POSIX_TRACE_RESUME shall mark the end of a trace overflow condition.
- If the Trace Log option is supported, POSIX_TRACE_FLUSH_START shall mark the beginning of a flush operation.

- If the Trace Log option is supported, POSIX_TRACE_FLUSH_STOP shall mark the end of a flush operation.
- If an implementation-defined trace error condition is reported, it shall be marked POSIX_TRACE_ERROR.

The interpretation of a trace stream or a trace log by a trace analyzer process relies on the information recorded for each trace event, and also on system trace events that indicate the invocation of trace control operations and trace system operational trace events.

The POSIX_TRACE_START and POSIX_TRACE_STOP trace events specify the time windows during which the trace stream is running.

The POSIX_TRACE_STOP trace event with an associated data that is equal to zero indicates a call of the function *posix_trace_stop*( ).

The POSIX_TRACE_STOP trace event with an associated data that is different from zero indicates an automatic stop of the trace stream (see *posix_trace_attr_getstreamfullpolicy*( ) defined in the System Interfaces volume of IEEE Std 1003.1-200x).

The POSIX_TRACE_FILTER trace event indicates that a trace event type filter value changed while the trace stream was running.

The POSIX_TRACE_ERROR serves to inform the analyzer process that an implementation-defined internal error of the trace system occurred.

The POSIX_TRACE_OVERFLOW trace event shall be reported with a timestamp equal to the timestamp of the first trace event overwritten. This is an indication that some generated trace events have been lost.

The POSIX_TRACE_RESUME trace event shall be reported with a timestamp equal to the timestamp of the first valid trace event reported after the overflow condition ends and shall be reported before this first valid trace event. This is an indication that the trace system is reliably recording trace events after an overflow condition.

Each of these trace event types shall be defined by a constant trace event name and a **trace_event_id_t** constant; trace event data is associated with some of these trace events.

If the Trace option is supported and the Trace Event Filter option and the Trace Log option are not supported, the following predefined system trace events in Table 2-3 shall be defined:

Table 2-3 Trace Option: System Trace Events

Event Name	Constant	Associated Data Type
posix_trace_error	POSIX_TRACE_ERROR	error
		int
posix_trace_start	POSIX_TRACE_START	None.
posix_trace_stop	POSIX_TRACE_STOP	auto
		int
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.

If the Trace option and the Trace Event Filter option are both supported, and if the Trace Log option is not supported, the following predefined system trace events in Table 2-4 (on page 528) shall be defined:

Table 2-4 Trace and Trace Event Filter Options: System Trace Events

Event Name	Constant	Associated Data Data Type
posix_trace_error	POSIX_TRACE_ERROR	error int
posix_trace_start	POSIX_TRACE_START	event_filter trace_event_set_t
posix_trace_stop	POSIX_TRACE_STOP	auto int
posix_trace_filter	POSIX_TRACE_FILTER	old_event_filter new_event_filter trace_event_set_t
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.

If the Trace option and the Trace Log option are both supported, and if the Trace Event Filter option is not supported, the following predefined system trace events in Table 2-5 shall be defined:

Table 2-5 Trace and Trace Log Options: System Trace Events

Event Name	Constant	Associated Data Data Type
posix_trace_error	POSIX_TRACE_ERROR	error - int
posix_trace_start	POSIX_TRACE_START	None.
posix_trace_stop	POSIX_TRACE_STOP	auto int
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.
posix_trace_flush_start	POSIX_TRACE_FLUSH_START	None.
posix_trace_flush_stop	POSIX_TRACE_FLUSH_STOP	None.

If the Trace option, the Trace Event Filter option, and the Trace Log option are all supported, the following predefined system trace events in Table 2-6 (on page 529) shall be defined:

Table 2-6 Trace, Trace Log, and Trace Event Filter Options: System Trace Events

Event Name	Constant	Associated Data Data Type
posix_trace_error	POSIX_TRACE_ERROR	error int
posix_trace_start	POSIX_TRACE_START	event_filter trace_event_set_t
posix_trace_stop	POSIX_TRACE_STOP	auto int
posix_trace_filter	POSIX_TRACE_FILTER	old_event_filter new_event_filter trace_event_set_t
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.
posix_trace_flush_start	POSIX_TRACE_FLUSH_START	None.
posix_trace_flush_stop	POSIX_TRACE_FLUSH_STOP	None.

2.11.2.2 *User Trace Event Type Definitions*

The user trace event POSIX_TRACE_UNNAMED_USEREVENT is defined in the **<trace.h>** header. If the limit of per-process user trace event names represented by {TRACE_USER_EVENT_MAX} has already been reached, this predefined user event shall be returned when the application tries to register more events than allowed. The data associated with this trace event is application-defined.

The following predefined user trace event in Table 2-7 shall be defined:

Table 2-7 Trace Option: User Trace Event

Event Name	Constant
posix_trace_unnamed_userevent	POSIX_TRACE_UNNAMED_USEREVENT

2.11.3 Trace Functions

The trace interface is built and structured to improve portability through use of trace data of opaque type. The object-oriented approach for the manipulation of trace attributes and trace event type identifiers requires definition of many constructor and selector functions which operate on these opaque types. Also, the trace interface must support several different tracing roles. To facilitate reading the trace interface, the trace functions are grouped into small functional sets supporting the three different roles:

- A trace controller process requires functions to set up and customize all the resources needed to run a trace stream, including:
  - — Attribute initialization and destruction (*posix_trace_attr_init*())
  - — Identification information manipulation (*posix_trace_attr_getgenversion*())
  - — Trace system behavior modification (*posix_trace_attr_getinherited*())
  - — Trace stream and trace log size set (*posix_trace_attr_getmaxusereventsize*())

- — Trace stream creation, flush, and shutdown (*posix_trace_create*( ))
- — Trace stream and trace log clear (*posix_trace_clear*( ))
- — Trace event type identifier manipulation (*posix_trace_trid_eventid_open*( ))
- — Trace event type identifier list exploration (*posix_trace_eventtypelist_getnext_id*( ))
- — Trace event type set manipulation (*posix_trace_eventset_empty*( ))
- — Trace event type filter set (*posix_trace_set_filter*( ))
- — Trace stream start and stop (*posix_trace_start*( ))
- — Trace stream information and status read (*posix_trace_get_attr*( ))

- A traced process requires functions to instrument trace points:
  - — Trace event type identifiers definition and trace points insertion (*posix_trace_event*( ))
- A trace analyzer process requires functions to retrieve information from a trace stream and trace log:
  - — Identification information read (*posix_trace_attr_getgenversion*( ))
  - — Trace system behavior information read (*posix_trace_attr_getinherited*( ))
  - — Trace stream and trace log size get (*posix_trace_attr_getmaxusereventsize*( ))
  - — Trace event type identifier manipulation (*posix_trace_trid_eventid_open*( ))
  - — Trace event type identifier list exploration (*posix_trace_eventtypelist_getnext_id*( ))
  - — Trace log open, rewind, and close (*posix_trace_open*( ))
  - — Trace stream information and status read (*posix_trace_get_attr*( ))
  - — trace event read (*posix_trace_getnext_event*( ))

2.12 Data Types

All of the data types used by various functions are defined by the implementation. The following table describes some of these types. Other types referenced in the description of a function, not mentioned here, can be found in the appropriate header for that function.

Defined Type	Description
cc_t	Type used for terminal special characters.
clock_t	Arithmetic type used for processor times, as defined in the ISO C standard.
clockid_t	Used for clock ID type in some timer functions.
dev_t	Arithmetic type used for device numbers.
DIR	Type representing a directory stream.
div_t	Structure type returned by the *div*( ) function.
FILE	Structure containing information about a file.
glob_t	Structure type used in pathname pattern matching.
fpos_t	Type containing all information needed to specify uniquely every

Defined Type	Description
	position within a file.
gid_t	Arithmetic type used for group IDs.
iconv_t	Type used for conversion descriptors.
id_t	Arithmetic type used as a general identifier; can be used to contain at least the largest of a **pid_t**, **uid_t**, or **gid_t**.
ino_t	Arithmetic type used for file serial numbers.
key_t	Arithmetic type used for XSI interprocess communication.
ldiv_t	Structure type returned by the *ldiv*( ) function.
mode_t	Arithmetic type used for file attributes.
mqd_t	Used for message queue descriptors.
nfds_t	Integer type used for the number of file descriptors.
nlink_t	Arithmetic type used for link counts.
off_t	Signed arithmetic type used for file sizes.
pid_t	Signed arithmetic type used for process and process group IDs.
pthread_attr_t	Used to identify a thread attribute object.
pthread_cond_t	Used for condition variables.
pthread_condattr_t	Used to identify a condition attribute object.
pthread_key_t	Used for thread-specific data keys.
pthread_mutex_t	Used for mutexes.
pthread_mutexattr_t	Used to identify a mutex attribute object.
pthread_once_t	Used for dynamic package initialization.
pthread_rwlock_t	Used for read-write locks.
pthread_rwlockattr_t	Used for read-write lock attributes.
pthread_t	Used to identify a thread.
ptrdiff_t	Signed integer type of the result of subtracting two pointers.
regex_t	Structure type used in regular expression matching.
regmatch_t	Structure type used in regular expression matching.
rlim_t	Unsigned arithmetic type used for limit values, to which objects of type **int** and **off_t** can be cast without loss of value.
sem_t	Type used in performing semaphore operations.
sig_atomic_t	Integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.
sigset_t	Integer or structure type of an object used to represent sets of signals.
size_t	Unsigned integer type used for size of objects.
speed_t	Type used for terminal baud rates.
ssize_t	Arithmetic type used for a count of bytes or an error indication.
suseconds_t	Signed arithmetic type used for time in microseconds.
tcflag_t	Type used for terminal modes.
time_t	Arithmetic type used for time in seconds, as defined in the ISO C standard.
timer_t	Used for timer ID returned by the *timer_create*( ) function.
uid_t	Arithmetic type used for user IDs.
useconds_t	Integer type used for time in microseconds.
va_list	Type used for traversing variable argument lists.
wchar_t	Integer type whose range of values can represent distinct codes for all members of the largest extended character set specified by the

Defined Type	Description
	supported locales.
wctype_t	Scalar type which represents a character class descriptor.
wint_t	Integer type capable of storing any valid value of **wchar_t** or WEOF.
wordexp_t	Structure type used in word expansion.

Chapter 3

System Interfaces

This chapter describes the functions, macros, and external variables to support applications portability at the C-language source level.

3345 **NAME**

3346 FD_CLR — macros for synchronous I/O multiplexing

3347 **SYNOPSIS**

3348 ```
#include <sys/time.h>
```

3349 ```
FD_CLR(int fd, fd_set *fdset);
```
3350 ```
FD_ISSET(int fd, fd_set *fdset);
```
3351 ```
FD_SET(int fd, fd_set *fdset);
```
3352 ```
FD_ZERO(fd_set *fdset);
```

3353 **DESCRIPTION**

3354 Refer to *pselect*( ).
```

3355 **NAME**

3356 _Exit, _exit — terminate a process

3357 **SYNOPSIS**

3358
```
#include <unistd.h>
```

3359
```
void _Exit(int status);
```
3360
```
void _exit(int status);
```

3361 **DESCRIPTION**

3362 Refer to *exit*( ).

3363 **NAME**

3364 _longjmp, _setjmp — non-local goto

3365 **SYNOPSIS**

3366 XSI
```
#include <setjmp.h>

void _longjmp(jmp_buf env, int val);
int _setjmp(jmp_buf env);
```
3367
3368
3369

3370 **DESCRIPTION**

3371 The *_longjmp*() and *_setjmp*() functions shall be equivalent to *longjmp*() and *setjmp*(), |
3372 respectively, with the additional restriction that *_longjmp*() and *_setjmp*() shall not manipulate |
3373 the signal mask. |

3374 If *_longjmp*() is called even though *env* was never initialized by a call to *_setjmp*(), or when the
3375 last such call was in a function that has since returned, the results are undefined.

3376 **RETURN VALUE**

3377 Refer to *longjmp*() and *setjmp*().

3378 **ERRORS**

3379 No errors are defined.

3380 **EXAMPLES**

3381 None.

3382 **APPLICATION USAGE**

3383 If *_longjmp*() is executed and the environment in which *_setjmp*() was executed no longer exists,
3384 errors can occur. The conditions under which the environment of the *_setjmp*() no longer exists
3385 include exiting the function that contains the *_setjmp*() call, and exiting an inner block with
3386 temporary storage. This condition might not be detectable, in which case the *_longjmp*() occurs
3387 and, if the environment no longer exists, the contents of the temporary storage of an inner block
3388 are unpredictable. This condition might also cause unexpected process termination. If the
3389 function has returned, the results are undefined.

3390 Passing *longjmp*() a pointer to a buffer not created by *setjmp*(), passing *_longjmp*() a pointer to a
3391 buffer not created by *_setjmp*(), passing *siglongjmp*() a pointer to a buffer not created by
3392 *sigsetjmp*(), or passing any of these three functions a buffer that has been modified by the user
3393 can cause all the problems listed above, and more.

3394 The *_longjmp*() and *_setjmp*() functions are included to support programs written to historical
3395 system interfaces. New applications should use *siglongjmp*() and *sigsetjmp*() respectively.

3396 **RATIONALE**

3397 None.

3398 **FUTURE DIRECTIONS**

3399 The *_longjmp*() and *_setjmp*() functions may be marked LEGACY in a future version.

3400 **SEE ALSO**

3401 *longjmp*(), *setjmp*(), *siglongjmp*(), *sigsetjmp*(), the Base Definitions volume of
3402 IEEE Std 1003.1-200x, **<setjmp.h>**

3403 **CHANGE HISTORY**

3404 First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

3407 **NAME**

3408 _setjmp — set jump point for a non-local goto

3409 **SYNOPSIS**

3410 XSI
```
#include <setjmp.h>
```

3411
```
int _setjmp(jmp_buf env);
```

3412

3413 **DESCRIPTION**

3414 Refer to *_longjmp*( ).

3415 **NAME**

3416 _tolower — transliterate uppercase characters to lowercase

3417 **SYNOPSIS**

3418 XSI
```
#include <ctype.h>
```

3419
```
int _tolower(int c);
```

3420

3421 **DESCRIPTION**

3422 The *_tolower*( ) macro shall be equivalent to *tolower*(*c*) except that the application shall ensure
3423 that the argument *c* is an uppercase letter.

3424 **RETURN VALUE**

3425 Upon successful completion, *_tolower*( ) shall return the lowercase letter corresponding to the
3426 argument passed.

3427 **ERRORS**

3428 No errors are defined.

3429 **EXAMPLES**

3430 None.

3431 **APPLICATION USAGE**

3432 None.

3433 **RATIONALE**

3434 None.

3435 **FUTURE DIRECTIONS**

3436 None.

3437 **SEE ALSO**

3438 *tolower*( ), *isupper*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base
3439 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

3440 **CHANGE HISTORY**

3441 First released in Issue 1. Derived from Issue 1 of the SVID.

3442 **Issue 6**

3443 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

_toupper — transliterate lowercase characters to uppercase

SYNOPSIS

XSI
```
#include <ctype.h>

int _toupper(int c);
```

DESCRIPTION

The *_toupper*( ) macro shall be equivalent to *toupper*( ) except that the application shall ensure that the argument *c* is a lowercase letter.

RETURN VALUE

Upon successful completion, *_toupper*( ) shall return the uppercase letter corresponding to the argument passed.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

islower( ), *toupper*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

a64l, l64a — convert between a 32-bit integer and a radix-64 ASCII string

SYNOPSIS

XSI

```
#include <stdlib.h>

long a64l(const char *s);
char *l64a(long value);
```

DESCRIPTION

These functions maintain numbers stored in radix-64 ASCII characters. This is a notation by which 32-bit integers can be represented by up to six characters; each character represents a digit in radix-64 notation. If the type **long** contains more than 32 bits, only the low-order 32 bits shall be used for these operations.

The characters used to represent digits are `'.'` (dot) for 0, `'/'` for 1, `'0'` through `'9'` for [2,11], `'A'` through `'Z'` for [12,37], and `'a'` through `'z'` for [38,63].

The *a64l*( ) function shall take a pointer to a radix-64 representation, in which the first digit is the least significant, and return the corresponding **long** value. If the string pointed to by *s* contains more than six characters, *a64l*( ) shall use the first six. If the first six characters of the string contain a null terminator, *a64l*( ) shall use only characters preceding the null terminator. The *a64l*( ) function shall scan the character string from left to right with the least significant digit on the left, decoding each character as a 6-bit radix-64 number. If the type **long** contains more than 32 bits, the resulting value is sign-extended. The behavior of *a64l*( ) is unspecified if *s* is a null pointer or the string pointed to by *s* was not generated by a previous call to *l64a*( ).

The *l64a*( ) function shall take a **long** argument and return a pointer to the corresponding radix-64 representation. The behavior of *l64a*( ) is unspecified if *value* is negative.

The value returned by *l64a*( ) may be a pointer into a static buffer. Subsequent calls to *l64a*( ) may overwrite the buffer.

The *l64a*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *a64l*( ) shall return the **long** value resulting from conversion of the input string. If a string pointed to by *s* is an empty string, *a64l*( ) shall return 0L.

The *l64a*( ) function shall return a pointer to the radix-64 representation. If *value* is 0L, *l64a*( ) shall return a pointer to an empty string.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

If the type **long** contains more than 32 bits, the result of *a64l*(*l64a*(*x*)) is *x* in the low-order 32 bits.

RATIONALE

This is not the same encoding as used by either encoding variant of the *uuencode* utility.

FUTURE DIRECTIONS

None.

SEE ALSO

strtoul( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *uuencode*

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section moved to the DESCRIPTION.

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

NAME

abort — generate an abnormal process abort

SYNOPSIS

```
#include <stdlib.h>

void abort(void);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *abort*() function shall cause abnormal process termination to occur, unless the signal SIGABRT is being caught and the signal handler does not return.

CX The abnormal termination processing shall include at least the effect of *fclose*() on all open streams and the default actions defined for SIGABRT.

XSI On XSI-conformant systems, in addition the abnormal termination processing shall include the effect of *fclose*() on message catalog descriptors.

The SIGABRT signal shall be sent to the calling process as if by means of *raise*() with the argument SIGABRT.

CX The status made available to *wait*() or *waitpid*() by *abort*() shall be that of a process terminated by the SIGABRT signal. The *abort*() function shall override blocking or ignoring the SIGABRT signal.

RETURN VALUE

The *abort*() function shall not return.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Catching the signal is intended to provide the application writer with a portable means to abort processing, free from possible interference from any implementation-defined functions.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exit(), *kill*(), *raise*(), *signal*(), *wait*(), *waitpid*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

abs — return an integer absolute value

SYNOPSIS

```
#include <stdlib.h>

int abs(int i);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *abs*( ) function shall compute the absolute value of its integer operand, *i*. If the result cannot be represented, the behavior is undefined.

RETURN VALUE

The *abs*( ) function shall return the absolute value of its integer operand.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

In two's-complement representation, the absolute value of the negative integer with largest magnitude {INT_MIN} might not be representable.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fabs( ), *labs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

3595 **NAME**

3596 accept — accept a new connection on a socket

3597 **SYNOPSIS**

3598 ```
#include <sys/socket.h>
```

3599 ```
int accept(int socket, struct sockaddr *restrict address,
```
3600 ```
    socklen_t *restrict address_len);
```

3601 **DESCRIPTION**

3602 The *accept*( ) function shall extract the first connection on the queue of pending connections,
3603 create a new socket with the same socket type protocol and address family as the specified
3604 socket, and allocate a new file descriptor for that socket.

3605 The *accept*( ) function takes the following arguments:

3606 *socket* Specifies a socket that was created with *socket*( ), has been bound to an address
3607 with *bind*( ), and has issued a successful call to *listen*( ).

3608 *address* Either a null pointer, or a pointer to a **sockaddr** structure where the address of
3609 the connecting socket shall be returned.

3610 *address_len* Points to a **socklen_t** structure which on input specifies the length of the
3611 supplied **sockaddr** structure, and on output specifies the length of the stored
3612 address.

3613 If *address* is not a null pointer, the address of the peer for the accepted connection shall be stored
3614 in the **sockaddr** structure pointed to by *address*, and the length of this address shall be stored in
3615 the object pointed to by *address_len*.

3616 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
3617 the stored address shall be truncated.

3618 If the protocol permits connections by unbound clients, and the peer is not bound, then the value
3619 stored in the object pointed to by *address* is unspecified.

3620 If the listen queue is empty of connection requests and O_NONBLOCK is not set on the file
3621 descriptor for the socket, *accept*( ) shall block until a connection is present. If the *listen*( ) queue is
3622 empty of connection requests and O_NONBLOCK is set on the file descriptor for the socket,
3623 *accept*( ) shall fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

3624 The accepted socket cannot itself accept more connections. The original socket remains open and
3625 can accept more connections.

3626 **RETURN VALUE**

3627 Upon successful completion, *accept*( ) shall return the non-negative file descriptor of the accepted
3628 socket. Otherwise, –1 shall be returned and *errno* set to indicate the error.

3629 **ERRORS**

3630 The *accept*( ) function shall fail if:

3631 [EAGAIN] or [EWOULDBLOCK]
3632 O_NONBLOCK is set for the socket file descriptor and no connections are
3633 present to be accepted.

3634 [EBADF] The *socket* argument is not a valid file descriptor.

3635 [ECONNABORTED]
3636 A connection has been aborted.
```

[EINTR] The *accept*( ) function was interrupted by a signal that was caught before a valid connection arrived.

[EINVAL] The *socket* is not accepting connections.

[EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

[ENFILE] The maximum number of file descriptors in the system are already open.

[ENOTSOCK] The *socket* argument does not refer to a socket.

[EOPNOTSUPP] The socket type of the specified socket does not support accepting connections.

The *accept*( ) function may fail if:

[ENOBUFS] No buffer space is available.

[ENOMEM] There was insufficient memory available to complete the operation.

XSR [EPROTO] A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized.

EXAMPLES

None.

APPLICATION USAGE

When a connection is available, *select*( ) indicates that the file descriptor for the socket is ready for reading.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

bind( ), *connect*( ), *listen*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the *accept*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

3666 **NAME**

3667 access — determine accessibility of a file

3668 **SYNOPSIS**

3669 ```
#include <unistd.h>
```

3670 ```
int access(const char *path, int amode);
```

3671 **DESCRIPTION**

3672 The *access*( ) function shall check the file named by the pathname pointed to by the *path* |
3673 argument for accessibility according to the bit pattern contained in *amode*, using the real user ID
3674 in place of the effective user ID and the real group ID in place of the effective group ID.

3675 The value of *amode* is either the bitwise-inclusive OR of the access permissions to be checked
3676 (R_OK, W_OK, X_OK) or the existence test (F_OK).

3677 If any access permissions are checked, each shall be checked individually, as described in the
3678 Base Definitions volume of IEEE Std 1003.1-200x, Chapter 3, Definitions. If the process has
3679 appropriate privileges, an implementation may indicate success for X_OK even if none of the
3680 execute file permission bits are set.

3681 **RETURN VALUE**

3682 If the requested access is permitted, *access*( ) succeeds and shall return 0; otherwise, –1 shall be
3683 returned and *errno* shall be set to indicate the error.

3684 **ERRORS**

3685 The *access*( ) function shall fail if:

3686 [EACCES] Permission bits of the file mode do not permit the requested access, or search
3687 permission is denied on a component of the path prefix.

3688 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
3689 argument.

3690 [ENAMETOOLONG]
3691 The length of the *path* argument exceeds {PATH_MAX} or a pathname |
3692 component is longer than {NAME_MAX}. |

3693 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

3694 [ENOTDIR] A component of the path prefix is not a directory.

3695 [EROFS] Write access is requested for a file on a read-only file system.

3696 The *access*( ) function may fail if:

3697 [EINVAL] The value of the *amode* argument is invalid.

3698 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
3699 resolution of the *path* argument.

3700 [ENAMETOOLONG]
3701 As a result of encountering a symbolic link in resolution of the *path* argument, |
3702 the length of the substituted pathname string exceeded {PATH_MAX}. |

3703 [ETXTBSY] Write access is requested for a pure procedure (shared text) file that is being
3704 executed.

EXAMPLES

Testing for the Existence of a File

The following example tests whether a file named **myfile** exists in the **/tmp** directory.

```
#include <unistd.h>
...
int result;
const char *filename = "/tmp/myfile";

result = access (filename, F_OK);
```

APPLICATION USAGE

Additional values of *amode* other than the set defined in the description may be valid; for example, if a system has extended access controls.

RATIONALE

In early proposals, some inadequacies in the *access*( ) function led to the creation of an *eaccess*( ) function because:

1. Historical implementations of *access*( ) do not test file access correctly when the process' real user ID is superuser. In particular, they always return zero when testing execute permissions without regard to whether the file is executable.

2. The superuser has complete access to all files on a system. As a consequence, programs started by the superuser and switched to the effective user ID with lesser privileges cannot use *access*( ) to test their file access permissions.

However, the historical model of *eaccess*( ) does not resolve problem (1), so this volume of IEEE Std 1003.1-200x now allows *access*( ) to behave in the desired way because several implementations have corrected the problem. It was also argued that problem (2) is more easily solved by using *open*( ), *chdir*( ), or one of the *exec* functions as appropriate and responding to the error, rather than creating a new function that would not be as reliable. Therefore, *eaccess*( ) is not included in this volume of IEEE Std 1003.1-200x.

The sentence concerning appropriate privileges and execute permission bits reflects the two possibilities implemented by historical implementations when checking superuser access for X_OK. |

New implementations are discouraged from returning X_OK unless at least one execution permission bit is set. | |

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *stat*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

- The [ETXTBSY] optional error condition is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

3750 **NAME**

3751 acos, acosf, acosl — arc cosine functions

3752 **SYNOPSIS**

3753
```
#include <math.h>
```

3754
```
double acos(double x);
```
3755
```
float acosf(float x);
```
3756
```
long double acosl(long double x);
```

3757 **DESCRIPTION**

3758 CX The functionality described on this reference page is aligned with the ISO C standard. Any
3759 conflict between the requirements described here and the ISO C standard is unintentional. This
3760 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

3761 These functions shall compute the principal value of the arc cosine of their argument *x*. The
3762 value of *x* should be in the range [–1,1].

3763 An application wishing to check for error situations should set *errno* to zero and call
3764 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
3765 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
3766 zero, an error has occurred.

3767 **RETURN VALUE**

3768 Upon successful completion, these functions shall return the arc cosine of *x*, in the range [0,π]
3769 radians.

3770 MX For finite values of *x* not in the range [–1,1], a domain error shall occur, and either a NaN (if
3771 supported), or an implementation-defined value shall be returned.

3772 MX If *x* is NaN, a NaN shall be returned.

3773 If *x* is +1, +0 shall be returned.

3774 If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-
3775 defined value shall be returned.

3776 **ERRORS**

3777 These functions shall fail if:

3778 MX Domain Error The *x* argument is finite and is not in the range [–1,1], or is ±Inf.

3779 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
3780 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling |
3781 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception |
3782 shall be raised. |

3783 **EXAMPLES**

3784 None.

3785 **APPLICATION USAGE**

3786 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
3787 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

3788 **RATIONALE**

3789 None.

FUTURE DIRECTIONS

None.

SEE ALSO

cos( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *acosf*( ) and *acosl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

3806 **NAME**

3807 acosf — arc cosine functions

3808 **SYNOPSIS**

3809 `#include <math.h>`

3810 `float acosf(float x);`

3811 **DESCRIPTION**

3812 Refer to *acos*( ).

NAME

acosh, acoshf, acoshl, — inverse hyperbolic cosine functions

SYNOPSIS

```
#include <math.h>

double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the inverse hyperbolic cosine of their argument *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the inverse hyperbolic cosine of their argument.

MX For finite values of $x < 1$, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is +1, +0 shall be returned.

If *x* is +Inf, +Inf shall be returned.

If *x* is –Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS

These functions shall fail if:

MX Domain Error — The *x* argument is finite and less than +1.0, or is –Inf.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

3853 **FUTURE DIRECTIONS**

3854 None.

3855 **SEE ALSO**

3856 *cosh*( ), *feclearexcept*( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section |
3857 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

3858 **CHANGE HISTORY**

3859 First released in Issue 4, Version 2.

3860 **Issue 5**

3861 Moved from X/OPEN UNIX extension to BASE.

3862 **Issue 6**

3863 The *acosh*( ) function is no longer marked as an extension.

3864 The *acoshf*( ), and *acoshl*( ) functions are added for alignment with the ISO/IEC 9899:1999
3865 standard.

3866 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
3867 revised to align with the ISO/IEC 9899:1999 standard.

3868 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
3869 marked.

3870 **NAME**

3871 acosl — arc cosine functions

3872 **SYNOPSIS**

3873 `#include <math.h>`

3874 `long double acosl(long double` *x*`);`

3875 **DESCRIPTION**

3876 Refer to *acos*( ).

NAME

aio_cancel — cancel an asynchronous I/O request (**REALTIME**)

SYNOPSIS

AIO

```
#include <aio.h>

int aio_cancel(int fildes, struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_cancel*( ) function shall attempt to cancel one or more asynchronous I/O requests currently outstanding against file descriptor *fildes*. The *aiocbp* argument points to the asynchronous I/O control block for a particular request to be canceled. If *aiocbp* is NULL, then all outstanding cancelable asynchronous I/O requests against *fildes* shall be canceled.

Normal asynchronous notification shall occur for asynchronous I/O operations that are successfully canceled. If there are requests that cannot be canceled, then the normal asynchronous completion process shall take place for those requests when they are completed.

For requested operations that are successfully canceled, the associated error status shall be set to [ECANCELED] and the return status shall be –1. For requested operations that are not successfully canceled, the *aiocbp* shall not be modified by *aio_cancel*( ).

If *aiocbp* is not NULL, then if *fildes* does not have the same value as the file descriptor with which the asynchronous operation was initiated, unspecified results occur.

Which operations are cancelable is implementation-defined.

RETURN VALUE

The *aio_cancel*( ) function shall return the value AIO_CANCELED to the calling process if the requested operation(s) were canceled. The value AIO_NOTCANCELED shall be returned if at least one of the requested operation(s) cannot be canceled because it is in progress. In this case, the state of the other operations, if any, referenced in the call to *aio_cancel*( ) is not indicated by the return value of *aio_cancel*( ). The application may determine the state of affairs for these operations by using *aio_error*( ). The value AIO_ALLDONE is returned if all of the operations have already completed. Otherwise, the function shall return –1 and set *errno* to indicate the error.

ERRORS

The *aio_cancel*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

EXAMPLES

None.

APPLICATION USAGE

The *aio_cancel*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

3918 **SEE ALSO**

3919 *aio_read*( ), *aio_write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

3920 **CHANGE HISTORY**

3921 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

3922 **Issue 6**

3923 The [ENOSYS] error condition has been removed as stubs need not be provided if an
3924 implementation does not support the Asynchronous Input and Output option.

3925 The APPLICATION USAGE section is added.

NAME

aio_error — retrieve errors status for an asynchronous I/O operation (**REALTIME**)

SYNOPSIS

AIO
```
#include <aio.h>

int aio_error(const struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_error*( ) function shall return the error status associated with the **aiocb** structure referenced by the *aiocbp* argument. The error status for an asynchronous I/O operation is the *errno* value that would be set by the corresponding *read*( ), *write*( ), *fdatasync*( ), or *fsync*( ) operation. If the operation has not yet completed, then the error status shall be equal to [EINPROGRESS].

RETURN VALUE

If the asynchronous I/O operation has completed successfully, then 0 shall be returned. If the asynchronous operation has completed unsuccessfully, then the error status, as described for *read*( ), *write*( ), *fdatasync*( ), and *fsync*( ), shall be returned. If the asynchronous I/O operation has not yet completed, then [EINPROGRESS] shall be returned.

ERRORS

The *aio_error*( ) function may fail if:

[EINVAL] The *aiocbp* argument does not refer to an asynchronous operation whose return status has not yet been retrieved.

EXAMPLES

None.

APPLICATION USAGE

The *aio_error*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel( ), *aio_fsync*( ), *aio_read*( ), *aio_return*( ), *aio_write*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lio_listio*( ), *lseek*( ), *read*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

NAME

aio_fsync — asynchronous file synchronization (**REALTIME**)

SYNOPSIS

AIO

```
#include <aio.h>

int aio_fsync(int op, struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_fsync*( ) function shall asynchronously force all I/O operations associated with the file indicated by the file descriptor *aio_fildes* member of the **aiocb** structure referenced by the *aiocbp* argument and queued at the time of the call to *aio_fsync*( ) to the synchronized I/O completion state. The function call shall return when the synchronization request has been initiated or queued to the file or device (even when the data cannot be synchronized immediately).

If *op* is O_DSYNC, all currently queued I/O operations shall be completed as if by a call to *fdatasync*( ); that is, as defined for synchronized I/O data integrity completion. If *op* is O_SYNC, all currently queued I/O operations shall be completed as if by a call to *fsync*( ); that is, as defined for synchronized I/O file integrity completion. If the *aio_fsync*( ) function fails, or if the operation queued by *aio_fsync*( ) fails, then, as for *fsync*( ) and *fdatasync*( ), outstanding I/O operations are not guaranteed to have been completed.

If *aio_fsync*( ) succeeds, then it is only the I/O that was queued at the time of the call to *aio_fsync*( ) that is guaranteed to be forced to the relevant completion state. The completion of subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronized fashion.

The *aiocbp* argument refers to an asynchronous I/O control block. The *aiocbp* value may be used as an argument to *aio_error*( ) and *aio_return*( ) in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. When the request is queued, the error status for the operation is [EINPROGRESS]. When all data has been successfully transferred, the error status shall be reset to reflect the success or failure of the operation. If the operation does not complete successfully, the error status for the operation shall be set to indicate the error. The *aio_sigevent* member determines the asynchronous notification to occur as specified in Section 2.4.1 (on page 478) when all operations have achieved synchronized I/O completion. All other members of the structure referenced by *aiocbp* are ignored. If the control block referenced by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

If the *aio_fsync*( ) function fails or the *aiocbp* indicates an error condition, data is not guaranteed to have been successfully transferred.

RETURN VALUE

The *aio_fsync*( ) function shall return the value 0 to the calling process if the I/O operation is successfully queued; otherwise, the function shall return the value –1 and set *errno* to indicate the error.

ERRORS

The *aio_fsync*( ) function shall fail if:

[EAGAIN] The requested asynchronous operation was not queued due to temporary resource limitations.

[EBADF] The *aio_fildes* member of the **aiocb** structure referenced by the *aiocbp* argument is not a valid file descriptor open for writing.

[EINVAL] This implementation does not support synchronized I/O for this file.

[EINVAL] A value of *op* other than O_DSYNC or O_SYNC was specified.

In the event that any of the queued I/O operations fail, *aio_fsync*() shall return the error condition defined for *read*() and *write*(). The error is returned in the error status for the asynchronous *fsync*() operation, which can be retrieved using *aio_error*().

EXAMPLES

None.

APPLICATION USAGE

The *aio_fsync*() function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fcntl(), *fdatasync*(), *fsync*(), *open*(), *read*(), *write*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

NAME

aio_read — asynchronous read from a file (**REALTIME**)

SYNOPSIS

AIO
```
#include <aio.h>

int aio_read(struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_read*() function shall read *aiocbp->aio_nbytes* from the file associated with *aiocbp->aio_fildes* into the buffer pointed to by *aiocbp->aio_buf*. The function call shall return when the read request has been initiated or queued to the file or device (even when the data cannot be delivered immediately).

PIO If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted at a priority equal to the scheduling priority of the process minus *aiocbp->aio_reqprio*.

The *aiocbp* value may be used as an argument to *aio_error*() and *aio_return*() in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. If an error condition is encountered during queuing, the function call shall return without having initiated or queued the request. The requested operation takes place at the absolute position in the file as given by *aio_offset*, as if *lseek*() were called immediately prior to the operation with an *offset* equal to *aio_offset* and a *whence* equal to SEEK_SET. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The *aiocbp->aio_lio_opcode* field shall be ignored by *aio_read*().

The *aiocbp* argument points to an **aiocb** structure. If the buffer pointed to by *aiocbp->aio_buf* or the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

SIO If synchronized I/O is enabled on the file associated with *aiocbp->aio_fildes*, the behavior of this function shall be according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding to the address range being changed, the result of that action is undefined.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with *aiocbp->aio_fildes*.

RETURN VALUE

The *aio_read*() function shall return the value zero to the calling process if the I/O operation is successfully queued; otherwise, the function shall return the value –1 and set *errno* to indicate the error.

ERRORS

The *aio_read*() function shall fail if:

[EAGAIN] The requested asynchronous I/O operation was not queued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to *aio_read*(), or asynchronously. If any of the conditions below are detected synchronously, the *aio_read*() function shall return –1 and set *errno* to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation

is set to –1, and the error status of the asynchronous operation is set to the corresponding value.

[EBADF] The *aiocbp->aio_fildes* argument is not a valid file descriptor open for reading.

[EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid, *aiocbp->aio_reqprio* is not a valid value, or *aiocbp->aio_nbytes* is an invalid value.

In the case that the *aio_read*( ) successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operation is one of the values normally returned by the *read*( ) function call. In addition, the error status of the asynchronous operation is set to one of the error statuses normally set by the *read*( ) function call, or one of the following values:

[EBADF] The *aiocbp->aio_fildes* argument is not a valid file descriptor open for reading.

[ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit *aio_cancel*( ) request.

[EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid.

The following condition may be detected synchronously or asynchronously:

[EOVERFLOW] The file is a regular file, *aiobcp->aio_nbytes* is greater than 0, and the starting offset in *aiobcp->aio_offset* is before the end-of-file and is at or beyond the offset maximum in the open file description associated with *aiocbp->aio_fildes*.

EXAMPLES

None.

APPLICATION USAGE

The *aio_read*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel( ), *aio_error*( ), *lio_listio*( ), *aio_return*( ), *aio_write*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lseek*( ), *read*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.
- In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support large files.

NAME

aio_return — retrieve return status of an asynchronous I/O operation (**REALTIME**)

SYNOPSIS

AIO

```
#include <aio.h>

ssize_t aio_return(struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_return*( ) function shall return the return status associated with the **aiocb** structure referenced by the *aiocbp* argument. The return status for an asynchronous I/O operation is the value that would be returned by the corresponding *read*( ), *write*( ), or *fsync*( ) function call. If the error status for the operation is equal to [EINPROGRESS], then the return status for the operation is undefined. The *aio_return*( ) function may be called exactly once to retrieve the return status of a given asynchronous operation; thereafter, if the same **aiocb** structure is used in a call to *aio_return*( ) or *aio_error*( ), an error may be returned. When the **aiocb** structure referred to by *aiocbp* is used to submit another asynchronous operation, then *aio_return*( ) may be successfully used to retrieve the return status of that operation.

RETURN VALUE

If the asynchronous I/O operation has completed, then the return status, as described for *read*( ), *write*( ), and *fsync*( ), shall be returned. If the asynchronous I/O operation has not yet completed, the results of *aio_return*( ) are undefined.

ERRORS

The *aio_return*( ) function may fail if:

[EINVAL] The *aiocbp* argument does not refer to an asynchronous operation whose return status has not yet been retrieved.

EXAMPLES

None.

APPLICATION USAGE

The *aio_return*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel( ), *aio_error*( ), *aio_fsync*( ), *aio_read*( ), *aio_write*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lio_listio*( ), *lseek*( ), *read*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

The [EINVAL] error condition is updated as a ‘‘may fail’’. This is for consistency with the DESCRIPTION.

NAME

aio_suspend — wait for an asynchronous I/O request (**REALTIME**)

SYNOPSIS

AIO
```
#include <aio.h>

int aio_suspend(const struct aiocb * const list[], int nent,
    const struct timespec *timeout);
```

DESCRIPTION

The *aio_suspend*( ) function shall suspend the calling thread until at least one of the asynchronous I/O operations referenced by the *list* argument has completed, until a signal interrupts the function, or, if *timeout* is not NULL, until the time interval specified by *timeout* has passed. If any of the **aiocb** structures in the list correspond to completed asynchronous I/O operations (that is, the error status for the operation is not equal to [EINPROGRESS]) at the time of the call, the function shall return without suspending the calling thread. The *list* argument is an array of pointers to asynchronous I/O control blocks. The *nent* argument indicates the number of elements in the array. Each **aiocb** structure pointed to has been used in initiating an asynchronous I/O request via *aio_read*( ), *aio_write*( ), or *lio_listio*( ). This array may contain NULL pointers, which are ignored. If this array contains pointers that refer to **aiocb** structures that have not been used in submitting asynchronous I/O, the effect is undefined.

If the time interval indicated in the **timespec** structure pointed to by *timeout* passes before any of the I/O operations referenced by *list* are completed, then *aio_suspend*( ) shall return with an error. MON If the Monotonic Clock option is supported, the clock that shall be used to measure this time interval shall be the CLOCK_MONOTONIC clock.

RETURN VALUE

If the *aio_suspend*( ) function returns after one or more asynchronous I/O operations have completed, the function shall return zero. Otherwise, the function shall return a value of –1 and set *errno* to indicate the error.

The application may determine which asynchronous I/O completed by scanning the associated error and return status using *aio_error*( ) and *aio_return*( ), respectively.

ERRORS

The *aio_suspend*( ) function shall fail if:

[EAGAIN] No asynchronous I/O indicated in the list referenced by *list* completed in the time interval indicated by *timeout*.

[EINTR] A signal interrupted the *aio_suspend*( ) function. Note that, since each asynchronous I/O operation may possibly provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited.

EXAMPLES

None.

APPLICATION USAGE

The *aio_suspend*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_read( ), *aio_write*( ), *lio_listio*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the CLOCK_MONOTONIC clock, if supported, is used.

NAME

aio_write — asynchronous write to a file (**REALTIME**)

SYNOPSIS

AIO
```
#include <aio.h>

int aio_write(struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_write*( ) function shall write *aiocbp->aio_nbytes* to the file associated with *aiocbp->aio_fildes* from the buffer pointed to by *aiocbp->aio_buf*. The function shall return when the write request has been initiated or, at a minimum, queued to the file or device.

PIO If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted at a priority equal to the scheduling priority of the process minus *aiocbp->aio_reqprio*.

The *aiocbp* may be used as an argument to *aio_error*( ) and *aio_return*( ) in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding.

The *aiocbp* argument points to an **aiocb** structure. If the buffer pointed to by *aiocbp->aio_buf* or the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

If O_APPEND is not set for the file descriptor *aio_fildes*, then the requested operation shall take place at the absolute position in the file as given by *aio_offset*, as if *lseek*( ) were called immediately prior to the operation with an *offset* equal to *aio_offset* and a *whence* equal to SEEK_SET. If O_APPEND is set for the file descriptor, write operations append to the file in the same order as the calls were made. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The *aiocbp->aio_lio_opcode* field shall be ignored by *aio_write*( ).

Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

SIO If synchronized I/O is enabled on the file associated with *aiocbp->aio_fildes*, the behavior of this function shall be according to the definitions of synchronized I/O data integrity completion, and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding to the address range being changed, the result of that action is undefined.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with *aiocbp->aio_fildes*.

RETURN VALUE

The *aio_write*( ) function shall return the value zero to the calling process if the I/O operation is successfully queued; otherwise, the function shall return –1 and set *errno* to indicate the error.

ERRORS

The *aio_write*( ) function shall fail if:

[EAGAIN] The requested asynchronous I/O operation was not queued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to *aio_write*( ), or asynchronously. If any of the conditions below are detected synchronously, the *aio_write*( ) function shall return –1 and set *errno* to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation

shall be set to –1, and the error status of the asynchronous operation is set to the corresponding value.

[EBADF] The *aiocbp->aio_fildes* argument is not a valid file descriptor open for writing.

[EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid, *aiocbp->aio_reqprio* is not a valid value, or *aiocbp->aio_nbytes* is an invalid value.

In the case that the *aio_write*( ) successfully queues the I/O operation, the return status of the asynchronous operation shall be one of the values normally returned by the *write*( ) function call. If the operation is successfully queued but is subsequently canceled or encounters an error, the error status for the asynchronous operation contains one of the values normally set by the *write*( ) function call, or one of the following:

[EBADF] The *aiocbp->aio_fildes* argument is not a valid file descriptor open for writing.

[EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid.

[ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit *aio_cancel*( ) request.

The following condition may be detected synchronously or asynchronously:

[EFBIG] The file is a regular file, *aiobcp->aio_nbytes* is greater than 0, and the starting offset in *aiobcp->aio_offset* is at or beyond the offset maximum in the open file description associated with *aiocbp->aio_fildes*.

EXAMPLES

None.

APPLICATION USAGE

The *aio_write*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel( ), *aio_error*( ), *aio_read*( ), *aio_return*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lio_listio*( ), *lseek*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text is added to indicate that for regular files no data transfer occurs past the offset maximum established in the open file description associated with *aiocbp->aio_fildes*.

- The [EFBIG] error is added as part of the large file support extensions.

4307 **NAME**

4308 alarm — schedule an alarm signal

4309 **SYNOPSIS**

4310
```
#include <unistd.h>
```

4311
```
unsigned alarm(unsigned seconds);
```

4312 **DESCRIPTION**

4313 The *alarm*( ) function shall cause the system to generate a SIGALRM signal for the process after
4314 the number of realtime seconds specified by *seconds* have elapsed. Processor scheduling delays
4315 may prevent the process from handling the signal as soon as it is generated.

4316 If *seconds* is 0, a pending alarm request, if any, is canceled.

4317 Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner.
4318 If the SIGALRM signal has not yet been generated, the call shall result in rescheduling the time
4319 at which the SIGALRM signal is generated.

4320 XSI Interactions between *alarm*( ) and any of *setitimer*( ), *ualarm*( ), or *usleep*( ) are unspecified.

4321 **RETURN VALUE**

4322 If there is a previous *alarm*( ) request with time remaining, *alarm*( ) shall return a non-zero value
4323 that is the number of seconds until the previous request would have generated a SIGALRM
4324 signal. Otherwise, *alarm*( ) shall return 0.

4325 **ERRORS**

4326 The *alarm*( ) function is always successful, and no return value is reserved to indicate an error.

4327 **EXAMPLES**

4328 None.

4329 **APPLICATION USAGE**

4330 The *fork*( ) function clears pending alarms in the child process. A new process image created by
4331 one of the *exec* functions inherits the time left to an alarm signal in the old process' image.

4332 Application writers should note that the type of the argument *seconds* and the return value of
4333 *alarm*( ) is **unsigned**. That means that a Strictly Conforming POSIX System Interfaces
4334 Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX},
4335 which the ISO C standard sets as 65 535, and any application passing a larger value is restricting
4336 its portability. A different type was considered, but historical implementations, including those
4337 with a 16-bit **int** type, consistently use either **unsigned** or **int**.

4338 Application writers should be aware of possible interactions when the same process uses both
4339 the *alarm*( ) and *sleep*( ) functions.

4340 **RATIONALE**

4341 Many historical implementations (including Version 7 and System V) allow an alarm to occur up
4342 to a second early. Other implementations allow alarms up to half a second or one clock tick
4343 early or do not allow them to occur early at all. The latter is considered most appropriate, since it
4344 gives the most predictable behavior, especially since the signal can always be delayed for an
4345 indefinite amount of time due to scheduling. Applications can thus choose the *seconds* argument
4346 as the minimum amount of time they wish to have elapse before the signal.

4347 The term *realtime* here and elsewhere (*sleep*( ), *times*( )) is intended to mean ''wall clock'' time as
4348 common English usage, and has nothing to do with ''realtime operating systems''. It is in
4349 contrast to *virtual time*, which could be misinterpreted if just *time* were used.

4350 In some implementations, including 4.3 BSD, very large values of the *seconds* argument are
4351 silently rounded down to an implementation-defined maximum value. This maximum is large

enough (on the order of several months) that the effect is not noticeable.

There were two possible choices for alarm generation in multi-threaded applications: generation for the calling thread or generation for the process. The first option would not have been particularly useful since the alarm state is maintained on a per-process basis and the alarm that is established by the last invocation of *alarm*( ) is the only one that would be active.

Furthermore, allowing generation of an asynchronous signal for a thread would have introduced an exception to the overall signal model. This requires a compelling reason in order to be justified.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *exec*, *fork*( ), *getitimer*( ), *pause*( ), *sigaction*( ), *sleep*( ), *ualarm*( ), *usleep*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to indicate that interactions with the *setitimer*( ), *ualarm*( ), and *usleep*( ) functions are unspecified.

NAME

asctime, asctime_r — convert date and time to a string

SYNOPSIS

```
#include <time.h>

char *asctime(const struct tm *timeptr);
TSF char *asctime_r(const struct tm *restrict tm, char *restrict buf);
```

DESCRIPTION

CX For *asctime*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *asctime*( ) function shall convert the broken-down time in the structure pointed to by *timeptr* into a string in the form:

```
Sun Sep 16 01:03:52 1973\n\0
```

using the equivalent of the following algorithm:

```
char *asctime(const struct tm *timeptr)
{
    static char wday_name[7][3] = {
        "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
    };
    static char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };
    static char result[26];

    sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
        wday_name[timeptr->tm_wday],
        mon_name[timeptr->tm_mon],
        timeptr->tm_mday, timeptr->tm_hour,
        timeptr->tm_min, timeptr->tm_sec,
        1900 + timeptr->tm_year);
    return result;
}
```

The **tm** structure is defined in the **<time.h>** header.

CX The *asctime*( ), *ctime*( ), *gmtime*( ), and *localtime*( ) functions shall return values in one of two static objects: a broken-down time structure and an array of type **char**. Execution of any of the functions may overwrite the information returned in either of these objects by any of the other functions.

The *asctime*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *asctime_r*( ) function shall convert the broken-down time in the structure pointed to by *tm* into a string (of the same form as that returned by *asctime*( )) that is placed in the user-supplied buffer pointed to by *buf* (which shall contain at least 26 bytes) and then return *buf*.

4415 **RETURN VALUE**

4416 Upon successful completion, *asctime*( ) shall return a pointer to the string.

4417 TSF Upon successful completion, *asctime_r*( ) shall return a pointer to a character string containing
4418 the date and time. This string is pointed to by the argument *buf*. If the function is unsuccessful,
4419 it shall return NULL.

4420 **ERRORS**

4421 No errors are defined.

4422 **EXAMPLES**

4423 None.

4424 **APPLICATION USAGE**

4425 Values for the broken-down time structure can be obtained by calling *gmtime*( ) or *localtime*( ).
4426 This function is included for compatibility with older implementations, and does not support
4427 localized date and time formats. Applications should use *strftime*( ) to achieve maximum
4428 portability.

4429 The *asctime_r*( ) function is thread-safe and shall return values in a user-supplied buffer instead
4430 of possibly using a static data area that may be overwritten by each call.

4431 **RATIONALE**

4432 None.

4433 **FUTURE DIRECTIONS**

4434 None.

4435 **SEE ALSO**

4436 *clock*( ), *ctime*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ),
4437 the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

4438 **CHANGE HISTORY**

4439 First released in Issue 1. Derived from Issue 1 of the SVID.

4440 **Issue 5**

4441 Normative text previously in the APPLICATION USAGE section is moved to the
4442 DESCRIPTION.

4443 The *asctime_r*( ) function is included for alignment with the POSIX Threads Extension.

4444 A note indicating that the *asctime*( ) function need not be reentrant is added to the
4445 DESCRIPTION.

4446 **Issue 6**

4447 The *asctime_r*( ) function is marked as part of the Thread-Safe Functions option.

4448 Extensions beyond the ISO C standard are now marked.

4449 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
4450 its avoidance of possibly using a static data area.

4451 The DESCRIPTION of *asctime_r*( ) is updated to describe the format of the string returned.

4452 The **restrict** keyword is added to the *asctime_r*( ) prototype for alignment with the
4453 ISO / IEC 9899: 1999 standard.

NAME

asin, asinf, asinl — arc sine function

SYNOPSIS

```
#include <math.h>

double asin(double x);
float asinf(float x);
long double asinl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the principal value of the arc sine of their argument *x*. The value of *x* should be in the range [–1,1].

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the arc sine of *x*, in the range $[-\pi/2,\pi/2]$ radians.

MX For finite values of *x* not in the range [–1,1], a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, *x* shall be returned.

If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

These functions shall fail if:

MX Domain Error — The *x* argument is finite and is not in the range [–1,1], or is ±Inf.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

These functions may fail if:

MX Range Error — The value of *x* is subnormal.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *isnan*( ), *sin*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *asinf*( ) and *asinl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

4517 **NAME**

4518 asinf — arc sine function

4519 **SYNOPSIS**

4520
```
#include <math.h>
```

4521
```
float asinf(float x);
```

4522 **DESCRIPTION**

4523 Refer to *asin*( ).

4524 **NAME**

4525 asinfh, asinfl — inverse hyperbolic sine functions

4526 **SYNOPSIS**

4527 `#include <math.h>`

4528 `float asinhf(float x);`

4529 `long double asinhl(long double x);`

4530 **DESCRIPTION**

4531 Refer to *asinh*( ).

NAME

asinh, asinfh, asinfl — inverse hyperbolic sine functions

SYNOPSIS

```
#include <math.h>

double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the inverse hyperbolic sine of their argument *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the inverse hyperbolic sine, of their argument.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, or ±Inf, *x* shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

These functions may fail if:

MX

Range Error	The value of *x* is subnormal.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *sinh*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *asinh*() function is no longer marked as an extension.

The *asinhf*(), and *asinhl*() functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

4585 **NAME**

4586 asinl — arc sine function

4587 **SYNOPSIS**

4588 `#include <math.h>`

4589 `long double asinl(long double x);`

4590 **DESCRIPTION**

4591 Refer to *asin*( ).

NAME

assert — insert program diagnostics

SYNOPSIS

```
#include <assert.h>

void assert(scalar expression);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *assert*( ) macro shall insert diagnostics into programs; it shall expand to a **void** expression. When it is executed, if *expression* (which shall have a **scalar** type) is false (that is, compares equal to 0), *assert*( ) shall write information about the particular call that failed on *stderr* and shall call *abort*( ).

The information written about the call that failed shall include the text of the argument, the name of the source file, the source file line number, and the name of the enclosing function, the latter are, respectively, the values of the preprocessing macros __FILE__ and __LINE__ and of the identifier __func__.

Forcing a definition of the name NDEBUG, either from the compiler command line or with the preprocessor control statement **#define** NDEBUG ahead of the **#include <assert.h>** statement, shall stop assertions from being compiled into the program.

RETURN VALUE

The *assert*( ) macro shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

abort( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<assert.h>**, *stderr*

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The prototype for the *expression* argument to *assert*( ) is changed from **int** to **scalar** for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION of *assert*( ) is updated for alignment with the ISO / IEC 9899: 1999 standard.

4632 **NAME**

4633 atan, atanf, atanl — arc tangent function

4634 **SYNOPSIS**

4635 `#include <math.h>`

4636 `double atan(double x);`
4637 `float atanf(float x);`
4638 `long double atanl(long double x);`

4639 **DESCRIPTION**

4640 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4641 conflict between the requirements described here and the ISO C standard is unintentional. This
4642 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

4643 These functions shall compute the principal value of the arc tangent of their argument *x*.

4644 An application wishing to check for error situations should set *errno* to zero and call
4645 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
4646 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
4647 zero, an error has occurred.

4648 **RETURN VALUE**

4649 Upon successful completion, these functions shall return the arc tangent of *x* in the range
4650 $[-\pi/2,\pi/2]$ radians.

4651 MX If *x* is NaN, a NaN shall be returned.

4652 If *x* is ±0 *x* shall be returned.

4653 If *x* is ±Inf, $\pm\pi/2$ shall be returned.

4654 If *x* is subnormal, a range error may occur and *x* should be returned.

4655 **ERRORS**

4656 These functions may fail if:

4657 MX Range Error The value of *x* is subnormal.

4658 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
4659 then *errno* shall be set to [ERANGE]. If the integer expression |
4660 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow |
4661 floating-point exception shall be raised. |

4662 **EXAMPLES**

4663 None.

4664 **APPLICATION USAGE**

4665 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
4666 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

4667 **RATIONALE**

4668 None.

4669 **FUTURE DIRECTIONS**

4670 None.

4671 **SEE ALSO**

4672 *atan2*( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *tan*( ), the Base Definitions volume of |
4673 IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, |
4674 **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *atanf*( ) and *atanl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

NAME

atan2, atan2f, atan2l — arc tangent functions

SYNOPSIS

```
#include <math.h>

double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the principal value of the arc tangent of y/x, using the signs of both arguments to determine the quadrant of the return value.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the arc tangent of y/x in the range $[-\pi,\pi]$ radians.

If y is ± 0 and x is < 0, $\pm\pi$ shall be returned.

If y is ± 0 and x is > 0, ± 0 shall be returned.

If y is < 0 and x is ± 0, $-\pi/2$ shall be returned.

If y is > 0 and x is ± 0, $\pi/2$ shall be returned.

If x is 0, a pole error shall not occur.

MX If either x or y is NaN, a NaN shall be returned.

If the result underflows, a range error may occur and y/x should be returned.

If y is ± 0 and x is -0, $\pm\pi$ shall be returned.

If y is ± 0 and x is $+0$, ± 0 shall be returned.

For finite values of $\pm y > 0$, if x is $-$Inf, $\pm\pi$ shall be returned.

For finite values of $\pm y > 0$, if x is $+$Inf, ± 0 shall be returned.

For finite values of x, if y is $\pm$Inf, $\pm\pi/2$ shall be returned.

If y is $\pm$Inf and x is $-$Inf, $\pm 3\pi/4$ shall be returned.

If y is $\pm$Inf and x is $+$Inf, $\pm\pi/4$ shall be returned.

If both arguments are 0, a domain error shall not occur.

ERRORS

These functions may fail if:

MX Range Error — The result underflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression

(math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

atan( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *tan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *atan2f*( ) and *atan2l*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

4753 **NAME**

4754 atanf — arc tangent function

4755 **SYNOPSIS**

4756
```
#include <math.h>
```

4757
```
float atanf(float x);
```

4758 **DESCRIPTION**

4759 Refer to *atan*( ).

NAME

atanh, atanhf, atanhl — inverse hyperbolic tangent functions

SYNOPSIS

```
#include <math.h>

double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the inverse hyperbolic tangent of their argument *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the inverse hyperbolic tangent of their argument.

If *x* is ±1, a pole error shall occur, and *atanh*( ), *atanhf*( ), and *atanhl*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively, with the same sign as the correct value of the function.

MX For finite $|x|>1$, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, *x* shall be returned.

If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

These functions shall fail if:

MX Domain Error — The *x* argument is finite and not in the range [–1,1], or is ±Inf.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

Pole Error — The *x* argument is ±1.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised.

These functions may fail if:

4802	MX	Range Error	The value of *x* is subnormal.
4803			If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

4807 **EXAMPLES**

4808 None.

4809 **APPLICATION USAGE**

4810 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
4811 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

4812 **RATIONALE**

4813 None.

4814 **FUTURE DIRECTIONS**

4815 None.

4816 **SEE ALSO**

4817 *feclearexcept*( ), *fetestexcept*( ), *tanh*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section
4818 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

4819 **CHANGE HISTORY**

4820 First released in Issue 4, Version 2.

4821 **Issue 5**

4822 Moved from X/OPEN UNIX extension to BASE.

4823 **Issue 6**

4824 The *atanh*( ) function is no longer marked as an extension.

4825 The *atanhf*( ), and *atanhl*( ) functions are added for alignment with the ISO/IEC 9899:1999
4826 standard.

4827 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
4828 revised to align with the ISO/IEC 9899:1999 standard.

4829 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
4830 marked.

4831 **NAME**

4832 atanl — arc tangent function

4833 **SYNOPSIS**

4834 `#include <math.h>`

4835 `long double atanl(long double x);`

4836 **DESCRIPTION**

4837 Refer to *atan*( ).

4838 **NAME**

4839 atexit — register a function to run at process termination

4840 **SYNOPSIS**

4841 `#include <stdlib.h>`

4842 `int atexit(void (*func)(void));`

4843 **DESCRIPTION**

4844 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4845 conflict between the requirements described here and the ISO C standard is unintentional. This
4846 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

4847 The *atexit*() function shall register the function pointed to by *func*, to be called without |
4848 arguments at normal program termination. At normal program termination, all functions
4849 registered by the *atexit*() function shall be called, in the reverse order of their registration, except
4850 that a function is called after any previously registered functions that had already been called at
4851 the time it was registered. Normal termination occurs either by a call to *exit*() or a return from
4852 *main*().

4853 At least 32 functions can be registered with *atexit*().

4854 CX After a successful call to any of the *exec* functions, any functions previously registered by *atexit*()
4855 shall no longer be registered.

4856 **RETURN VALUE**

4857 Upon successful completion, *atexit*() shall return 0; otherwise, it shall return a non-zero value.

4858 **ERRORS**

4859 No errors are defined.

4860 **EXAMPLES**

4861 None.

4862 **APPLICATION USAGE**

4863 The functions registered by a call to *atexit*() must return to ensure that all registered functions
4864 are called.

4865 The application should call *sysconf*() to obtain the value of {ATEXIT_MAX}, the number of
4866 functions that can be registered. There is no way for an application to tell how many functions
4867 have already been registered with *atexit*().

4868 **RATIONALE**

4869 None.

4870 **FUTURE DIRECTIONS**

4871 None.

4872 **SEE ALSO**

4873 *exit*(), *sysconf*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

4874 **CHANGE HISTORY**

4875 First released in Issue 4. Derived from the ANSI C standard.

4876 **Issue 6**

4877 Extensions beyond the ISO C standard are now marked.

4878 The DESCRIPTION is updated for alignment with the ISO/IEC 9899: 1999 standard.

4879 **NAME**

4880 atof — convert a string to double-precision number

4881 **SYNOPSIS**

4882 `#include <stdlib.h>`

4883 `double atof(const char *str);`

4884 **DESCRIPTION**

4885 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4886 conflict between the requirements described here and the ISO C standard is unintentional. This
4887 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

4888 The call *atof*(*str*) shall be equivalent to:

4889 `strtod(str,(char **)NULL),`

4890 except that the handling of errors may differ. If the value cannot be represented, the behavior is
4891 undefined.

4892 **RETURN VALUE**

4893 The *atof*( ) function shall return the converted value if the value can be represented.

4894 **ERRORS**

4895 No errors are defined.

4896 **EXAMPLES**

4897 None.

4898 **APPLICATION USAGE**

4899 The *atof*( ) function is subsumed by *strtod*( ) but is retained because it is used extensively in
4900 existing code. If the number is not known to be in range, *strtod*( ) should be used because *atof*( ) is
4901 not required to perform any error checking.

4902 **RATIONALE**

4903 None.

4904 **FUTURE DIRECTIONS**

4905 None.

4906 **SEE ALSO**

4907 *strtod*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

4908 **CHANGE HISTORY**

4909 First released in Issue 1. Derived from Issue 1 of the SVID.

4910 **NAME**

4911 atoi — convert a string to an integer

4912 **SYNOPSIS**

4913
```
#include <stdlib.h>
```

4914
```
int atoi(const char *str);
```

4915 **DESCRIPTION**

4916 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4917 conflict between the requirements described here and the ISO C standard is unintentional. This
4918 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

4919 The call *atoi*(*str*) shall be equivalent to:

4920
```
(int) strtol(str, (char **)NULL, 10)
```

4921 except that the handling of errors may differ. If the value cannot be represented, the behavior is
4922 undefined.

4923 **RETURN VALUE**

4924 The *atoi*( ) function shall return the converted value if the value can be represented.

4925 **ERRORS**

4926 No errors are defined.

4927 **EXAMPLES**

4928 **Converting an Argument**

4929 The following example checks for proper usage of the program. If there is an argument and the
4930 decimal conversion of this argument (obtained using *atoi*( )) is greater than 0, then the program
4931 has a valid number of minutes to wait for an event.

4932
```
#include <stdlib.h>
#include <stdio.h>
...
int minutes_to_event;
...
if (argc < 2 || ((minutes_to_event = atoi (argv[1]))) <= 0) {
   fprintf(stderr, "Usage: %s minutes\n", argv[0]); exit(1);
}
...
```

4941 **APPLICATION USAGE**

4942 The *atoi*( ) function is subsumed by *strtol*( ) but is retained because it is used extensively in
4943 existing code. If the number is not known to be in range, *strtol*( ) should be used because *atoi*( ) is
4944 not required to perform any error checking.

4945 **RATIONALE**

4946 None.

4947 **FUTURE DIRECTIONS**

4948 None.

4949 **SEE ALSO**

4950 *strtol*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

4951 **CHANGE HISTORY**

4952 First released in Issue 1. Derived from Issue 1 of the SVID.

4953 **NAME**

4954 atol, atoll — convert a string to a long integer

4955 **SYNOPSIS**

4956
```
#include <stdlib.h>
```

4957
```
long atol(const char *str);
```
4958
```
long long atoll(const char *nptr);
```

4959 **DESCRIPTION**

4960 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4961 conflict between the requirements described here and the ISO C standard is unintentional. This
4962 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

4963 The call *atol*(*str*) shall be equivalent to:

4964
```
strtol(str, (char **)NULL, 10)
```

4965 The call *atoll*(*str*) shall be equivalent to:

4966
```
strtoll(nptr, (char **)NULL, 10)
```

4967 except that the handling of errors may differ. If the value cannot be represented, the behavior is
4968 undefined.

4969 **RETURN VALUE**

4970 These functions shall return the converted value if the value can be represented.

4971 **ERRORS**

4972 No errors are defined.

4973 **EXAMPLES**

4974 None.

4975 **APPLICATION USAGE**

4976 The *atol*( ) function is subsumed by *strtol*( ) but is retained because it is used extensively in
4977 existing code. If the number is not known to be in range, *strtol*( ) should be used because *atol*( ) is
4978 not required to perform any error checking.

4979 **RATIONALE**

4980 None.

4981 **FUTURE DIRECTIONS**

4982 None.

4983 **SEE ALSO**

4984 *strtol*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

4985 **CHANGE HISTORY**

4986 First released in Issue 1. Derived from Issue 1 of the SVID.

4987 **Issue 6**

4988 The *atoll*( ) function is added for alignment with the ISO / IEC 9899: 1999 standard.

NAME

basename — return the last component of a pathname

SYNOPSIS

XSI

```
#include <libgen.h>

char *basename(char *path);
```

DESCRIPTION

The *basename*( ) function shall take the pathname pointed to by *path* and return a pointer to the final component of the pathname, deleting any trailing `'/'` characters.

If the string consists entirely of the `'/'` character, *basename*( ) shall return a pointer to the string `"/"`. If the string is exactly `"//"`, it is implementation-defined whether `'/'` or `"//"` is returned.

If *path* is a null pointer or points to an empty string, *basename*( ) shall return a pointer to the string `"."`.

The *basename*( ) function may modify the string pointed to by *path*, and may return a pointer to static storage that may then be overwritten by a subsequent call to *basename*( ).

The *basename*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *basename*( ) function shall return a pointer to the final component of *path*.

ERRORS

No errors are defined.

EXAMPLES

Using basename( )

The following program fragment returns a pointer to the value *lib*, which is the base name of **/usr/lib**.

```
#include <libgen.h>
...
char *name = "/usr/lib";
char *base;

base = basename(name);
...
```

Sample Input and Output Strings for basename( )

In the following table, the input string is the value pointed to by *path*, and the output string is the return value of the *basename*( ) function.

Input String	Output String
`"/usr/lib"`	`"lib"`
`"/usr/"`	`"usr"`
`"/"`	`"/"`

5028 **APPLICATION USAGE**

5029 None.

5030 **RATIONALE**

5031 None.

5032 **FUTURE DIRECTIONS**

5033 None.

5034 **SEE ALSO**

5035 *dirname*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<libgen.h>**, the Shell and
5036 Utilities volume of IEEE Std 1003.1-200x, *basename*

5037 **CHANGE HISTORY**

5038 First released in Issue 4, Version 2.

5039 **Issue 5**

5040 Moved from X/OPEN UNIX extension to BASE.

5041 Normative text previously in the APPLICATION USAGE section is moved to the
5042 DESCRIPTION.

5043 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

5044 **Issue 6**

5045 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

5046 **NAME**

5047 bcmp — memory operations (**LEGACY**)

5048 **SYNOPSIS**

5049 XSI
```
#include <strings.h>
```

5050
```
int bcmp(const void *s1, const void *s2, size_t n);
```

5051

5052 **DESCRIPTION**

5053 The *bcmp*( ) function shall compare the first *n* bytes of the area pointed to by *s1* with the area
5054 pointed to by *s2*.

5055 **RETURN VALUE**

5056 The *bcmp*( ) function shall return 0 if *s1* and *s2* are identical; otherwise, it shall return non-zero.
5057 Both areas are assumed to be *n* bytes long. If the value of *n* is 0, *bcmp*( ) shall return 0.

5058 **ERRORS**

5059 No errors are defined.

5060 **EXAMPLES**

5061 None.

5062 **APPLICATION USAGE**

5063 *memcmp*( ) is preferred over this function.

5064 For maximum portability, it is recommended to replace the function call to *bcmp*( ) as follows:

5065
```
#define bcmp(b1,b2,len) memcmp((b1), (b2), (size_t)(len))
```

5066 **RATIONALE**

5067 None.

5068 **FUTURE DIRECTIONS**

5069 This function may be withdrawn in a future version.

5070 **SEE ALSO**

5071 *memcmp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

5072 **CHANGE HISTORY**

5073 First released in Issue 4, Version 2.

5074 **Issue 5**

5075 Moved from X/OPEN UNIX extension to BASE.

5076 **Issue 6**

5077 This function is marked LEGACY.

NAME

bcopy — memory operations (**LEGACY**)

SYNOPSIS

XSI
```
#include <strings.h>

void bcopy(const void *s1, void *s2, size_t n);
```

DESCRIPTION

The *bcopy*( ) function shall copy *n* bytes from the area pointed to by *s1* to the area pointed to by *s2*.

The bytes are copied correctly even if the area pointed to by *s1* overlaps the area pointed to by *s2*.

RETURN VALUE

The *bcopy*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

memmove( ) is preferred over this function.

The following are approximately equivalent (note the order of the arguments):

```
bcopy(s1,s2,n) ~= memmove(s2,s1,n)
```

For maximum portability, it is recommended to replace the function call to *bcopy*( ) as follows:

```
#define bcopy(b1,b2,len) (memmove((b2), (b1), (len)), (void) 0)
```

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

memmove( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

5113 **NAME**

5114 bind — bind a name to a socket

5115 **SYNOPSIS**

5116 `#include <sys/socket.h>`

5117 `int bind(int socket, const struct sockaddr *address,`
5118 `socklen_t address_len);`

5119 **DESCRIPTION**

5120 The *bind*( ) function shall assign a local socket address *address* to a socket identified by descriptor
5121 *socket* that has no local socket address assigned. Sockets created with the *socket*( ) function are
5122 initially unnamed; they are identified only by their address family.

5123 The *bind*( ) function takes the following arguments:

5124 *socket* Specifies the file descriptor of the socket to be bound.

5125 *address* Points to a **sockaddr** structure containing the address to be bound to the
5126 socket. The length and format of the address depend on the address family of
5127 the socket.

5128 *address_len* Specifies the length of the **sockaddr** structure pointed to by the *address*
5129 argument.

5130 The socket specified by *socket* may require the process to have appropriate privileges to use the
5131 *bind*( ) function.

5132 **RETURN VALUE**

5133 Upon successful completion, *bind*( ) shall return 0; otherwise, –1 shall be returned and *errno* set
5134 to indicate the error.

5135 **ERRORS**

5136 The *bind*( ) function shall fail if:

5137 [EADDRINUSE]
5138 The specified address is already in use.

5139 [EADDRNOTAVAIL]
5140 The specified address is not available from the local machine.

5141 [EAFNOSUPPORT]
5142 The specified address is not a valid address for the address family of the
5143 specified socket.

5144 [EBADF] The *socket* argument is not a valid file descriptor.

5145 [EINVAL] The socket is already bound to an address, and the protocol does not support
5146 binding to a new address; or the socket has been shut down.

5147 [ENOTSOCK] The *socket* argument does not refer to a socket.

5148 [EOPNOTSUPP] The socket type of the specified socket does not support binding to an
5149 address.

5150 If the address family of the socket is AF_UNIX, then *bind*( ) shall fail if:

5151 [EACCES] A component of the path prefix denies search permission, or the requested
5152 name requires writing in a directory with a mode that denies write
5153 permission.

[EDESTADDRREQ] or [EISDIR]
The *address* argument is a null pointer.

[EIO] An I/O error occurred.

[ELOOP] A loop exists in symbolic links encountered during resolution of the pathname in *address*.

[ENAMETOOLONG]
A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.

[ENOENT] A component of the pathname does not name an existing file or the pathname is an empty string.

[ENOTDIR] A component of the path prefix of the pathname in *address* is not a directory.

[EROFS] The name would reside on a read-only file system.

The *bind*( ) function may fail if:

[EACCES] The specified address is protected and the current user does not have permission to bind to it.

[EINVAL] The *address_len* argument is not a valid length for the address family.

[EISCONN] The socket is already connected.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in *address*.

[ENAMETOOLONG]
Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

[ENOBUFS] Insufficient resources were available to complete the call.

EXAMPLES

None.

APPLICATION USAGE

An application program can retrieve the assigned socket name with the *getsockname*( ) function.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

connect( ), *getsockname*( ), *listen*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

bsd_signal — simplified signal facilities

SYNOPSIS

OB XSI

```
#include <signal.h>

void (*bsd_signal(int sig, void (*func)(int)))(int);
```

DESCRIPTION

The *bsd_signal*() function provides a partially compatible interface for programs written to historical system interfaces (see APPLICATION USAGE).

The function call *bsd_signal*(*sig*, *func*) shall be equivalent to the following:

```
void (*bsd_signal(int sig, void (*func)(int)))(int)
{
    struct sigaction act, oact;

    act.sa_handler = func;
    act.sa_flags = SA_RESTART;
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask, sig);
    if (sigaction(sig, &act, &oact) == -1)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

The handler function should be declared:

```
void handler(int sig);
```

where *sig* is the signal number. The behavior is undefined if *func* is a function that takes more than one argument, or an argument of a different type.

RETURN VALUE

Upon successful completion, *bsd_signal*() shall return the previous action for *sig*. Otherwise, SIG_ERR shall be returned and *errno* shall be set to indicate the error.

ERRORS

Refer to *sigaction*().

EXAMPLES

None.

APPLICATION USAGE

This function is a direct replacement for the BSD *signal*() function for simple applications that are installing a single-argument signal handler function. If a BSD signal handler function is being installed that expects more than one argument, the application has to be modified to use *sigaction*(). The *bsd_signal*() function differs from *signal*() in that the SA_RESTART flag is set and the SA_RESETHAND is clear when *bsd_signal*() is used. The state of these flags is not specified for *signal*().

It is recommended that new applications use the *sigaction*() function.

RATIONALE

None.

5232 **FUTURE DIRECTIONS**

5233 None.

5234 **SEE ALSO**

5235 *sigaction*( ), *sigaddset*( ), *sigemptyset*( ), *signal*( ), the Base Definitions volume of
5236 IEEE Std 1003.1-200x, **<signal.h>**

5237 **CHANGE HISTORY**

5238 First released in Issue 4, Version 2.

5239 **Issue 5**

5240 Moved from X/OPEN UNIX extension to BASE.

5241 **Issue 6**

5242 This function is marked obsolescent.

NAME

bsearch — binary search a sorted table

SYNOPSIS

```
#include <stdlib.h>

void *bsearch(const void *key, const void *base, size_t nel,
    size_t width, int (*compar)(const void *, const void *));
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *bsearch*( ) function shall search an array of *nel* objects, the initial element of which is pointed to by *base*, for an element that matches the object pointed to by *key*. The size of each element in the array is specified by *width*.

The comparison function pointed to by *compar* shall be called with two arguments that point to the *key* object and to an array element, in that order.

The application shall ensure that the function returns an integer less than, equal to, or greater than 0 if the *key* object is considered, respectively, to be less than, to match, or to be greater than the array element. The application shall ensure that the array consists of all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the *key* object, in that order.

RETURN VALUE

The *bsearch*( ) function shall return a pointer to a matching member of the array, or a null pointer if no match is found. If two or more members compare equal, which member is returned is unspecified.

ERRORS

No errors are defined.

EXAMPLES

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

The code fragment below reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TABSIZE    1000

struct node {                   /* These are stored in the table. */
    char *string;
    int length;
};
struct node table[TABSIZE];     /* Table to be searched. */
    .
    .
    .
{
    struct node *node_ptr, node;
    /* routine to compare 2 nodes */
```

```
        int node_compare(const void *, const void *);
        char str_space[20];   /* Space to read string into. */
        .
        .
        .
        node.string = str_space;
        while (scanf("%s", node.string) != EOF) {
            node_ptr = (struct node *)bsearch((void *)(&node),
                   (void *)table, TABSIZE,
                   sizeof(struct node), node_compare);
            if (node_ptr != NULL) {
                (void)printf("string = %20s, length = %d\n",
                    node_ptr->string, node_ptr->length);
            } else {
                (void)printf("not found: %s\n", node.string);
            }
        }
    }
    /*
        This routine compares two nodes based on an
        alphabetical ordering of the string field.
    */
    int
    node_compare(const void *node1, const void *node2)
    {
        return strcoll(((const struct node *)node1)->string,
            ((const struct node *)node2)->string);
    }
```

APPLICATION USAGE

The pointers to the key and the element at the base of the table should be of type pointer-to-element.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

In practice, the array is usually sorted according to the comparison function.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

hcreate( ), *lsearch*( ), *qsort*( ), *tsearch*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

5334 **NAME**

5335 btowc — single byte to wide character conversion

5336 **SYNOPSIS**

5337 ```
#include <stdio.h>
```

5338 ```
#include <wchar.h>
```

5339 ```
wint_t btowc(int c);
```

5340 **DESCRIPTION**

5341 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5342 conflict between the requirements described here and the ISO C standard is unintentional. This
5343 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5344 The *btowc*( ) function shall determine whether *c* constitutes a valid (one-byte) character in the
5345 initial shift state.

5346 The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

5347 **RETURN VALUE**

5348 The *btowc*( ) function shall return WEOF if *c* has the value EOF or if **(unsigned char)** *c* does not
5349 constitute a valid (one-byte) character in the initial shift state. Otherwise, it shall return the
5350 wide-character representation of that character.

5351 **ERRORS**

5352 No errors are defined.

5353 **EXAMPLES**

5354 None.

5355 **APPLICATION USAGE**

5356 None.

5357 **RATIONALE**

5358 None.

5359 **FUTURE DIRECTIONS**

5360 None.

5361 **SEE ALSO**

5362 *wctob*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

5363 **CHANGE HISTORY**

5364 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
5365 (E).
```

NAME

bzero — memory operations (**LEGACY**)

SYNOPSIS

XSI
```
#include <strings.h>

void bzero(void *s, size_t n);
```

DESCRIPTION

The *bzero*( ) function shall place *n* zero-valued bytes in the area pointed to by *s*.

RETURN VALUE

The *bzero*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

memset( ) is preferred over this function.

For maximum portability, it is recommended to replace the function call to *bzero*( ) as follows:

```
#define bzero(b,len) (memset((b), '\0', (len)), (void) 0)
```

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

memset( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

5396 **NAME**

5397 cabs, cabsf, cabsl — return a complex absolute value

5398 **SYNOPSIS**

5399
```
#include <complex.h>
```

5400
```
double cabs(double complex z);
```
5401
```
float cabsf(float complex z);
```
5402
```
long double cabsl(long double complex z);
```

5403 **DESCRIPTION**

5404 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5405 conflict between the requirements described here and the ISO C standard is unintentional. This
5406 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5407 These functions shall compute the complex absolute value (also called norm, modulus, or
5408 magnitude) of *z*.

5409 **RETURN VALUE**

5410 These functions shall return the complex absolute value.

5411 **ERRORS**

5412 No errors are defined.

5413 **EXAMPLES**

5414 None.

5415 **APPLICATION USAGE**

5416 None.

5417 **RATIONALE**

5418 None.

5419 **FUTURE DIRECTIONS**

5420 None.

5421 **SEE ALSO**

5422 The Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

5423 **CHANGE HISTORY**

5424 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

5425 **NAME**

5426 cacos, cacosf, cacosl — complex arc cosine functions

5427 **SYNOPSIS**

5428 ```
#include <complex.h>

double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
```

5432 **DESCRIPTION**

5433 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5434 conflict between the requirements described here and the ISO C standard is unintentional. This
5435 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5436 These functions shall compute the complex arc cosine of *z*, with branch cuts outside the interval
5437 [–1, +1] along the real axis.

5438 **RETURN VALUE**

5439 These functions shall return the complex arc cosine value, in the range of a strip mathematically
5440 unbounded along the imaginary axis and in the interval [0, π] along the real axis.

5441 **ERRORS**

5442 No errors are defined.

5443 **EXAMPLES**

5444 None.

5445 **APPLICATION USAGE**

5446 None.

5447 **RATIONALE**

5448 None.

5449 **FUTURE DIRECTIONS**

5450 None.

5451 **SEE ALSO**

5452 *ccos*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

5453 **CHANGE HISTORY**

5454 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.
```

5455 **NAME**

5456 cacosf — complex arc cosine functions

5457 **SYNOPSIS**

5458
```
#include <complex.h>
```

5459
```
float complex cacosf(float complex z);
```

5460 **DESCRIPTION**

5461 Refer to *cacos*( ).

5462 **NAME**

5463 cacosh, cacoshf, cacoshl — complex arc hyperbolic cosine functions

5464 **SYNOPSIS**

5465
```
#include <complex.h>
```

5466
```
double complex cacosh(double complex z);
```
5467
```
float complex cacoshf(float complex z);
```
5468
```
long double complex cacoshl(long double complex z);
```

5469 **DESCRIPTION**

5470 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5471 conflict between the requirements described here and the ISO C standard is unintentional. This
5472 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5473 These functions shall compute the complex arc hyperbolic cosine of *z*, with a branch cut at
5474 values less than 1 along the real axis.

5475 **RETURN VALUE**

5476 These functions shall return the complex arc hyperbolic cosine value, in the range of a half-strip
5477 of non-negative values along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

5478 **ERRORS**

5479 No errors are defined.

5480 **EXAMPLES**

5481 None.

5482 **APPLICATION USAGE**

5483 None.

5484 **RATIONALE**

5485 None.

5486 **FUTURE DIRECTIONS**

5487 None.

5488 **SEE ALSO**

5489 *ccosh*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

5490 **CHANGE HISTORY**

5491 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

5492 **NAME**

5493 cacosl — complex arc cosine functions

5494 **SYNOPSIS**

5495 `#include <complex.h>`

5496 `long double complex cacosl(long double complex z);`

5497 **DESCRIPTION**

5498 Refer to *cacos*( ).

NAME

calloc — a memory allocator

SYNOPSIS

```
#include <stdlib.h>

void *calloc(size_t nelem, size_t elsize);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *calloc*( ) function shall allocate unused space for an array of *nelem* elements each of whose size in bytes is *elsize*. The space shall be initialized to all bits 0.

The order and contiguity of storage allocated by successive calls to *calloc*( ) is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned shall point to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned. If the size of the space requested is 0, the behavior is implementation-defined: the value returned shall be either a null pointer or a unique pointer.

RETURN VALUE

Upon successful completion with both *nelem* and *elsize* non-zero, *calloc*( ) shall return a pointer to the allocated space. If either *nelem* or *elsize* is 0, then either a null pointer or a unique pointer value that can be successfully passed to *free*( ) shall be returned. Otherwise, it shall return a null pointer CX and set *errno* to indicate the error.

ERRORS

The *calloc*( ) function shall fail if:

CX [ENOMEM] Insufficient memory is available.

EXAMPLES

None.

APPLICATION USAGE

There is now no requirement for the implementation to support the inclusion of **<malloc.h>**.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

free( ), *malloc*( ), *realloc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The setting of *errno* and the [ENOMEM] error condition are mandatory if an insufficient memory condition occurs.

5544 **NAME**

5545 carg, cargf, cargl — complex argument functions

5546 **SYNOPSIS**

5547
```
#include <complex.h>
```

5548
```
double carg(double complex z);
```
5549
```
float cargf(float complex z);
```
5550
```
long double cargl(long double complex z);
```

5551 **DESCRIPTION**

5552 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5553 conflict between the requirements described here and the ISO C standard is unintentional. This
5554 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5555 These functions shall compute the argument (also called phase angle) of *z*, with a branch cut
5556 along the negative real axis.

5557 **RETURN VALUE**

5558 These functions shall return the value of the argument in the interval [–π, +π].

5559 **ERRORS**

5560 No errors are defined.

5561 **EXAMPLES**

5562 None.

5563 **APPLICATION USAGE**

5564 None.

5565 **RATIONALE**

5566 None.

5567 **FUTURE DIRECTIONS**

5568 None.

5569 **SEE ALSO**

5570 *cimag*( ), *conj*( ), *cproj*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

5571 **CHANGE HISTORY**

5572 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

casin, casinf, casinl — complex arc sine functions

SYNOPSIS

```
#include <complex.h>

double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex arc sine of *z*, with branch cuts outside the interval [–1, +1] along the real axis.

RETURN VALUE

These functions shall return the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

csin( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

5603 **NAME**

5604 casinf — complex arc sine functions

5605 **SYNOPSIS**

5606
```
#include <complex.h>
```

5607
```
float complex casinf(float complex z);
```

5608 **DESCRIPTION**

5609 Refer to *casin*( ).

NAME

casinh, casinhf, casinhl — complex arc hyperbolic sine functions

SYNOPSIS

```
#include <complex.h>

double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex arc hyperbolic sine of *z*, with branch cuts outside the interval [–*i*, +*i*] along the imaginary axis.

RETURN VALUE

These functions shall return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

csinh( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

5641 **NAME**

5642 casinl — complex arc sine functions

5643 **SYNOPSIS**

5644 `#include <complex.h>`

5645 `long double complex casinl(long double complex z);`

5646 **DESCRIPTION**

5647 Refer to *casin*( ).

NAME

catan, catanf, catanl — complex arc tangent functions

SYNOPSIS

```
#include <complex.h>

double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex arc tangent of *z*, with branch cuts outside the interval [–*i*, +*i*] along the imaginary axis.

RETURN VALUE

These functions shall return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [$-\pi/2$, $+\pi/2$] along the real axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ctan( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

5679 **NAME**

5680 catanf — complex arc tangent functions

5681 **SYNOPSIS**

5682 `#include <complex.h>`

5683 `float complex catanf(float complex` *z*`);`

5684 **DESCRIPTION**

5685 Refer to *catan*( ).

NAME

catanh, catanhf, catanhl — complex arc hyperbolic tangent functions

SYNOPSIS

```
#include <complex.h>

double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex arc hyperbolic tangent of z, with branch cuts outside the interval [–1, +1] along the real axis.

RETURN VALUE

These functions shall return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ctanh( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

5717 **NAME**

5718 catanl — complex arc tangent functions

5719 **SYNOPSIS**

5720 `#include <complex.h>`

5721 `long double complex catanl(long double complex z);`

5722 **DESCRIPTION**

5723 Refer to *catan*( ).

5724 **NAME**

5725 catclose — close a message catalog descriptor

5726 **SYNOPSIS**

5727 XSI
```
#include <nl_types.h>
```

5728
```
int catclose(nl_catd catd);
```

5729

5730 **DESCRIPTION**

5731 The *catclose*( ) function shall close the message catalog identified by *catd*. If a file descriptor is
5732 used to implement the type **nl_catd**, that file descriptor shall be closed.

5733 **RETURN VALUE**

5734 Upon successful completion, *catclose*( ) shall return 0; otherwise, –1 shall be returned, and *errno*
5735 set to indicate the error.

5736 **ERRORS**

5737 The *catclose*( ) function may fail if:

5738 [EBADF] The catalog descriptor is not valid.

5739 [EINTR] The *catclose*( ) function was interrupted by a signal.

5740 **EXAMPLES**

5741 None.

5742 **APPLICATION USAGE**

5743 None.

5744 **RATIONALE**

5745 None.

5746 **FUTURE DIRECTIONS**

5747 None.

5748 **SEE ALSO**

5749 *catgets*( ), *catopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<nl_types.h>**

5750 **CHANGE HISTORY**

5751 First released in Issue 2.

NAME

catgets — read a program message

SYNOPSIS

XSI
```
#include <nl_types.h>

char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);
```

DESCRIPTION

The *catgets*( ) function shall attempt to read message *msg_id*, in set *set_id*, from the message catalog identified by *catd*. The *catd* argument is a message catalog descriptor returned from an earlier call to *catopen*( ). The *s* argument points to a default message string which shall be returned by *catgets*( ) if it cannot retrieve the identified message.

The *catgets*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

If the identified message is retrieved successfully, *catgets*( ) shall return a pointer to an internal buffer area containing the null-terminated message string. If the call is unsuccessful for any reason, *s* shall be returned and *errno* may be set to indicate the error.

ERRORS

The *catgets*( ) function may fail if:

[EBADF] The *catd* argument is not a valid message catalog descriptor open for reading.

[EBADMSG] The message identified by *set_id* and *msg_id* in the specified message catalog did not satisfy implementation-defined security criteria.

[EINTR] The read operation was terminated due to the receipt of a signal, and no data was transferred.

[EINVAL] The message catalog identified by *catd* is corrupted.

[ENOMSG] The message identified by *set_id* and *msg_id* is not in the message catalog.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

catclose( ), *catopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<nl_types.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 5

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

NAME

catopen — open a message catalog

SYNOPSIS

XSI
```
#include <nl_types.h>

nl_catd catopen(const char *name, int oflag);
```

DESCRIPTION

The *catopen*( ) function shall open a message catalog and return a message catalog descriptor. The *name* argument specifies the name of the message catalog to be opened. If *name* contains a `'/'`, then *name* specifies a complete name for the message catalog. Otherwise, the environment variable *NLSPATH* is used with *name* substituted for the `%N` conversion specification (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables). If *NLSPATH* exists in the environment when the process starts, then if the process has appropriate privileges, the behavior of *catopen*( ) is undefined. If *NLSPATH* does not exist in the environment, or if a message catalog cannot be found in any of the components specified by *NLSPATH*, then an implementation-defined default path shall be used. This default may be affected by the setting of *LC_MESSAGES* if the value of *oflag* is NL_CAT_LOCALE, or the *LANG* environment variable if *oflag* is 0.

A message catalog descriptor shall remain valid in a process until that process closes it, or a successful call to one of the *exec* functions. A change in the setting of the *LC_MESSAGES* category may invalidate existing open catalogs.

If a file descriptor is used to implement message catalog descriptors, the FD_CLOEXEC flag shall be set; see **<fcntl.h>**.

If the value of the *oflag* argument is 0, the *LANG* environment variable is used to locate the catalog without regard to the *LC_MESSAGES* category. If the *oflag* argument is NL_CAT_LOCALE, the *LC_MESSAGES* category is used to locate the message catalog (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables).

RETURN VALUE

Upon successful completion, *catopen*( ) shall return a message catalog descriptor for use on subsequent calls to *catgets*( ) and *catclose*( ). Otherwise, *catopen*( ) shall return (**nl_catd**) –1 and set *errno* to indicate the error.

ERRORS

The *catopen*( ) function may fail if:

[EACCES] Search permission is denied for the component of the path prefix of the message catalog or read permission is denied for the message catalog.

[EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

[ENAMETOOLONG]
The length of a pathname of the message catalog exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

[ENAMETOOLONG]
Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

[ENFILE] Too many files are currently open in the system.

[ENOENT] The message catalog does not exist or the *name* argument points to an empty string.

[ENOMEM] Insufficient storage space is available.

[ENOTDIR] A component of the path prefix of the message catalog is not a directory.

EXAMPLES

None.

APPLICATION USAGE

Some implementations of *catopen*() use *malloc*() to allocate space for internal buffer areas. The *catopen*() function may fail if there is insufficient storage space available to accommodate these buffers.

Conforming applications must assume that message catalog descriptors are not valid after a call to one of the *exec* functions. |

Application writers should be aware that guidelines for the location of message catalogs have not yet been developed. Therefore they should take care to avoid conflicting with catalogs used by other applications and the standard utilities.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

catclose(), *catgets*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<nl_types.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 2.

5861 **NAME**

5862 cbrt, cbrtf, cbrtl — cube root functions

5863 **SYNOPSIS**

5864
```
#include <math.h>
```

5865
```
double cbrt(double x);
```
5866
```
float cbrtf(float x);
```
5867
```
long double cbrtl(long double x);
```

5868 **DESCRIPTION**

5869 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5870 conflict between the requirements described here and the ISO C standard is unintentional. This
5871 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5872 These functions shall compute the real cube root of their argument *x*.

5873 **RETURN VALUE**

5874 Upon successful completion, these functions shall return the cube root of *x*.

5875 MX If *x* is NaN, a NaN shall be returned.

5876 If *x* is ±0, or ±Inf, *x* shall be returned.

5877 **ERRORS**

5878 No errors are defined.

5879 **EXAMPLES**

5880 None.

5881 **APPLICATION USAGE**

5882 None.

5883 **RATIONALE**

5884 For some applications, a true cube root function, which returns negative results for negative
5885 arguments, is more appropriate than *pow*(*x*, 1.0/3.0), which returns a NaN for *x* less than 0.

5886 **FUTURE DIRECTIONS**

5887 None.

5888 **SEE ALSO**

5889 The Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

5890 **CHANGE HISTORY**

5891 First released in Issue 4, Version 2.

5892 **Issue 5**

5893 Moved from X/OPEN UNIX extension to BASE.

5894 **Issue 6**

5895 The *cbrt*( ) function is no longer marked as an extension.

5896 The *cbrtf*( ) and *cbrtl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

5897 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
5898 revised to align with the ISO/IEC 9899:1999 standard.

5899 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
5900 marked.

5901 **NAME**

5902 ccos, ccosf, ccosl — complex cosine functions

5903 **SYNOPSIS**

5904
```
#include <complex.h>
```

5905
```
double complex ccos(double complex z);
```
5906
```
float complex ccosf(float complex z);
```
5907
```
long double complex ccosl(long double complex z);
```

5908 **DESCRIPTION**

5909 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5910 conflict between the requirements described here and the ISO C standard is unintentional. This
5911 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5912 These functions shall compute the complex cosine of *z*.

5913 **RETURN VALUE**

5914 These functions shall return the complex cosine value.

5915 **ERRORS**

5916 No errors are defined.

5917 **EXAMPLES**

5918 None.

5919 **APPLICATION USAGE**

5920 None.

5921 **RATIONALE**

5922 None.

5923 **FUTURE DIRECTIONS**

5924 None.

5925 **SEE ALSO**

5926 *cacos*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

5927 **CHANGE HISTORY**

5928 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

ccosf — complex cosine functions

SYNOPSIS

```
#include <complex.h>

float complex ccosf(float complex z);
```

DESCRIPTION

Refer to *ccos*( ).

5936 **NAME**

5937 ccosh, ccoshf, ccoshl — complex hyperbolic cosine functions

5938 **SYNOPSIS**

5939
```
#include <complex.h>
```

5940
```
double complex ccosh(double complex z);
```
5941
```
float complex ccoshf(float complex z);
```
5942
```
long double complex ccoshl(long double complex z);
```

5943 **DESCRIPTION**

5944 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5945 conflict between the requirements described here and the ISO C standard is unintentional. This
5946 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5947 These functions shall compute the complex hyperbolic cosine of *z*.

5948 **RETURN VALUE**

5949 These functions shall return the complex hyperbolic cosine value.

5950 **ERRORS**

5951 No errors are defined.

5952 **EXAMPLES**

5953 None.

5954 **APPLICATION USAGE**

5955 None.

5956 **RATIONALE**

5957 None.

5958 **FUTURE DIRECTIONS**

5959 None.

5960 **SEE ALSO**

5961 *cacosh*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

5962 **CHANGE HISTORY**

5963 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

5964 **NAME**

5965 ccosl — complex cosine functions

5966 **SYNOPSIS**

5967
```
#include <complex.h>
```

5968
```
long double complex ccosl(long double complex z);
```

5969 **DESCRIPTION**

5970 Refer to *ccos*( ).

5971 **NAME**

5972 ceil, ceilf, ceill — ceiling value function

5973 **SYNOPSIS**

5974 `#include <math.h>`

5975 `double ceil(double x);`
5976 `float ceilf(float x);`
5977 `long double ceill(long double x);`

5978 **DESCRIPTION**

5979 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5980 conflict between the requirements described here and the ISO C standard is unintentional. This
5981 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5982 These functions shall compute the smallest integral value not less than *x*.

5983 An application wishing to check for error situations should set *errno* to zero and call
5984 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
5985 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
5986 zero, an error has occurred.

5987 **RETURN VALUE**

5988 Upon successful completion, *ceil*( ), *ceilf*( ), and *ceill*( ) shall return the smallest integral value not
5989 less than *x*, expressed as a type **double**, **float**, or **long double**, respectively.

5990 MX If *x* is NaN, a NaN shall be returned.

5991 If *x* is ±0, or ±Inf, *x* shall be returned.

5992 XSI If the correct value would cause overflow, a range error shall occur and *ceil*( ), *ceilf*( ), and *ceill*( )
5993 shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

5994 **ERRORS**

5995 These functions shall fail if:

5996 XSI

Range Error	The result overflows.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

5997 5998 5999 6000

6001 **EXAMPLES**

6002 None.

6003 **APPLICATION USAGE**

6004 The integral value returned by these functions need not be expressible as an **int** or **long**. The
6005 return value should be tested before assigning it to an integer type to avoid the undefined results
6006 of an integer overflow.

6007 The *ceil*( ) function can only overflow when the floating-point representation has
6008 DBL_MANT_DIG > DBL_MAX_EXP.

6009 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
6010 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *floor*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *ceilf*( ) and *ceill*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

6029 **NAME**

6030 cexp, cexpf, cexpl — complex exponential functions

6031 **SYNOPSIS**

6032
```
#include <complex.h>
```

6033
```
double complex cexp(double complex z);
```
6034
```
float complex cexpf(float complex z);
```
6035
```
long double complex cexpl(long double complex z);
```

6036 **DESCRIPTION**

6037 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6038 conflict between the requirements described here and the ISO C standard is unintentional. This
6039 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

6040 These functions shall compute the complex exponent of z, defined as e^z.

6041 **RETURN VALUE**

6042 These functions shall return the complex exponential value of z.

6043 **ERRORS**

6044 No errors are defined.

6045 **EXAMPLES**

6046 None.

6047 **APPLICATION USAGE**

6048 None.

6049 **RATIONALE**

6050 None.

6051 **FUTURE DIRECTIONS**

6052 None.

6053 **SEE ALSO**

6054 *clog*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

6055 **CHANGE HISTORY**

6056 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

6057 **NAME**

6058 cfgetispeed — get input baud rate

6059 **SYNOPSIS**

6060 `#include <termios.h>`

6061 `speed_t cfgetispeed(const struct termios *termios_p);`

6062 **DESCRIPTION**

6063 The *cfgetispeed*( ) function shall extract the input baud rate from the **termios** structure to which
6064 the *termios_p* argument points.

6065 This function shall return exactly the value in the **termios** data structure, without interpretation.

6066 **RETURN VALUE**

6067 Upon successful completion, *cfgetispeed*( ) shall return a value of type **speed_t** representing the
6068 input baud rate.

6069 **ERRORS**

6070 No errors are defined.

6071 **EXAMPLES**

6072 None.

6073 **APPLICATION USAGE**

6074 None.

6075 **RATIONALE**

6076 The term *baud* is used historically here, but is not technically correct. This is properly ‘‘bits per
6077 second’’, which may not be the same as baud. However, the term is used because of the
6078 historical usage and understanding.

6079 The *cfgetospeed*( ), *cfgetispeed*( ), *cfsetospeed*( ), and *cfsetispeed*( ) functions do not take arguments as
6080 numbers, but rather as symbolic names. There are two reasons for this:

6081 1. Historically, numbers were not used because of the way the rate was stored in the data
6082 structure. This is retained even though a function is now used.

6083 2. More importantly, only a limited set of possible rates is at all portable, and this constrains
6084 the application to that set.

6085 There is nothing to prevent an implementation to accept, as an extension, a number (such as 126)
6086 if it wished, and because the encoding of the Bxxx symbols is not specified, this can be done so
6087 no ambiguity is introduced.

6088 Setting the input baud rate to zero was a mechanism to allow for split baud rates. Clarifications
6089 in this volume of IEEE Std 1003.1-200x have made it possible to determine whether split rates are
6090 supported and to support them without having to treat zero as a special case. Since this
6091 functionality is also confusing, it has been declared obsolescent. The 0 argument referred to is
6092 the literal constant 0, not the symbolic constant B0. This volume of IEEE Std 1003.1-200x does
6093 not preclude B0 from being defined as the value 0; in fact, implementations would likely benefit
6094 from the two being equivalent. This volume of IEEE Std 1003.1-200x does not fully specify
6095 whether the previous *cfsetispeed*( ) value is retained after a *tcgetattr*( ) as the actual value or as |
6096 zero. Therefore, conforming applications should always set both the input speed and output |
6097 speed when setting either.

6098 In historical implementations, the baud rate information is traditionally kept in **c_cflag**.
6099 Applications should be written to presume that this might be the case (and thus not blindly copy
6100 **c_cflag**), but not to rely on it in case it is in some other field of the structure. Setting the **c_cflag**
6101 field absolutely after setting a baud rate is a non-portable action because of this. In general, the

unused parts of the flag fields might be used by the implementation and should not be blindly copied from the descriptions of one terminal device to another.

FUTURE DIRECTIONS

None.

SEE ALSO

cfgetospeed( ), *cfsetispeed*( ), *cfsetospeed*( ), *tcgetattr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

6113 **NAME**

6114 cfgetospeed — get output baud rate

6115 **SYNOPSIS**

6116 `#include <termios.h>`

6117 `speed_t cfgetospeed(const struct termios *termios_p);`

6118 **DESCRIPTION**

6119 The *cfgetospeed*( ) function shall extract the output baud rate from the **termios** structure to which
6120 the *termios_p* argument points.

6121 This function shall return exactly the value in the **termios** data structure, without interpretation.

6122 **RETURN VALUE**

6123 Upon successful completion, *cfgetospeed*( ) shall return a value of type **speed_t** representing the
6124 output baud rate.

6125 **ERRORS**

6126 No errors are defined.

6127 **EXAMPLES**

6128 None.

6129 **APPLICATION USAGE**

6130 None.

6131 **RATIONALE**

6132 Refer to *cfgetispeed*( ).

6133 **FUTURE DIRECTIONS**

6134 None.

6135 **SEE ALSO**

6136 *cfgetispeed*( ), *cfsetispeed*( ), *cfsetospeed*( ), *tcgetattr*( ), the Base Definitions volume of
6137 IEEE Std 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std 1003.1-200x,
6138 Chapter 11, General Terminal Interface

6139 **CHANGE HISTORY**

6140 First released in Issue 3.

6141 Entry included for alignment with the POSIX.1-1988 standard.

NAME

cfsetispeed — set input baud rate

SYNOPSIS

```
#include <termios.h>

int cfsetispeed(struct termios *termios_p, speed_t speed);
```

DESCRIPTION

The *cfsetispeed*( ) function shall set the input baud rate stored in the structure pointed to by *termios_p* to *speed*.

There shall be no effect on the baud rates set in the hardware until a subsequent successful call to *tcsetattr*( ) with the same **termios** structure. Similarly, errors resulting from attempts to set baud rates not supported by the terminal device need not be detected until the *tcsetattr*( ) function is called.

RETURN VALUE

Upon successful completion, *cfsetispeed*( ) shall return 0; otherwise, −1 shall be returned, and *errno* may be set to indicate the error.

ERRORS

The *cfsetispeed*( ) function may fail if:

[EINVAL] The *speed* value is not a valid baud rate.

[EINVAL] The value of *speed* is outside the range of possible speed values as specified in **<termios.h>**.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to *cfgetispeed*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

cfgetispeed( ), *cfgetospeed*( ), *cfsetospeed*( ), *tcsetattr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The optional setting of *errno* and the [EINVAL] error conditions are added.

NAME

cfsetospeed — set output baud rate

SYNOPSIS

```
#include <termios.h>

int cfsetospeed(struct termios *termios_p, speed_t speed);
```

DESCRIPTION

The *cfsetospeed*( ) function shall set the output baud rate stored in the structure pointed to by *termios_p* to *speed*.

There shall be no effect on the baud rates set in the hardware until a subsequent successful call to *tcsetattr*( ) with the same **termios** structure. Similarly, errors resulting from attempts to set baud rates not supported by the terminal device need not be detected until the *tcsetattr*( ) function is called.

RETURN VALUE

Upon successful completion, *cfsetospeed*( ) shall return 0; otherwise, it shall return –1 and *errno* may be set to indicate the error.

ERRORS

The *cfsetospeed*( ) function may fail if:

[EINVAL] The *speed* value is not a valid baud rate.

[EINVAL] The value of *speed* is outside the range of possible speed values as specified in **<termios.h>**.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to *cfgetispeed*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

cfgetispeed( ), *cfgetospeed*( ), *cfsetispeed*( ), *tcsetattr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The optional setting of *errno* and the [EINVAL] error conditions are added.

NAME

chdir — change working directory

SYNOPSIS

```
#include <unistd.h>

int chdir(const char *path);
```

DESCRIPTION

The *chdir*( ) function shall cause the directory named by the pathname pointed to by the *path* argument to become the current working directory; that is, the starting point for path searches for pathnames not beginning with '//'.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned, the current working directory shall remain unchanged, and *errno* shall be set to indicate the error.

ERRORS

The *chdir*( ) function shall fail if:

[EACCES] Search permission is denied for any component of the pathname.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

[ENOENT] A component of *path* does not name an existing directory or *path* is an empty string.

[ENOTDIR] A component of the pathname is not a directory.

The *chdir*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG] As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

EXAMPLES

Changing the Current Working Directory

The following example makes the value pointed to by **directory**, **/tmp**, the current working directory.

```
#include <unistd.h>
...
char *directory = "/tmp";
int ret;

ret = chdir (directory);
```

6258 **APPLICATION USAGE**

6259 None.

6260 **RATIONALE**

6261 The *chdir*( ) function only affects the working directory of the current process.

6262 **FUTURE DIRECTIONS**

6263 None.

6264 **SEE ALSO**

6265 *getcwd*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

6266 **CHANGE HISTORY**

6267 First released in Issue 1. Derived from Issue 1 of the SVID.

6268 **Issue 6**

6269 The APPLICATION USAGE section is added.

6270 The following new requirements on POSIX implementations derive from alignment with the | 6271 Single UNIX Specification:

6272 • The [ELOOP] mandatory error condition is added.

6273 • A second [ENAMETOOLONG] is added as an optional error condition.

6274 The following changes were made to align with the IEEE P1003.1a draft standard:

6275 • The [ELOOP] optional error condition is added.

NAME

chmod — change mode of a file

SYNOPSIS

```
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
```

DESCRIPTION

XSI The *chmod*( ) function shall change S_ISUID, S_ISGID, S_ISVTX, and the file permission bits of the file named by the pathname pointed to by the *path* argument to the corresponding bits in the *mode* argument. The application shall ensure that the effective user ID of the process matches the owner of the file or the process has appropriate privileges in order to do this.

XSI S_ISUID, S_ISGID, S_ISVTX, and the file permission bits are described in **<sys/stat.h>**.

If the calling process does not have appropriate privileges, and if the group ID of the file does not match the effective group ID or one of the supplementary group IDs and if the file is a regular file, bit S_ISGID (set-group-ID on execution) in the file's mode shall be cleared upon successful return from *chmod*( ).

Additional implementation-defined restrictions may cause the S_ISUID and S_ISGID bits in *mode* to be ignored.

The effect on file descriptors for files open at the time of a call to *chmod*( ) is implementation-defined.

Upon successful completion, *chmod*( ) shall mark for update the *st_ctime* field of the file.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to indicate the error. If –1 is returned, no change to the file mode occurs.

ERRORS

The *chmod*( ) function shall fail if:

[EACCES] Search permission is denied on a component of the path prefix.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

[EPERM] The effective user ID does not match the owner of the file and the process does not have appropriate privileges.

[EROFS] The named file resides on a read-only file system.

The *chmod*( ) function may fail if:

[EINTR] A signal was caught during execution of the function.

[EINVAL] The value of the *mode* argument is invalid.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname strings exceeded {PATH_MAX}.

EXAMPLES

Setting Read Permissions for User, Group, and Others

The following example sets read permissions for the owner, group, and others.

```
#include <sys/stat.h>

const char *path;
...
chmod(path, S_IRUSR|S_IRGRP|S_IROTH);
```

Setting Read, Write, and Execute Permissions for the Owner Only

The following example sets read, write, and execute permissions for the owner, and no permissions for group and others.

```
#include <sys/stat.h>

const char *path;
...
chmod(path, S_IRWXU);
```

Setting Different Permissions for Owner, Group, and Other

The following example sets owner permissions for CHANGEFILE to read, write, and execute, group permissions to read and execute, and other permissions to read.

```
#include <sys/stat.h>

#define CHANGEFILE "/etc/myfile"
...
chmod(CHANGEFILE, S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH);
```

Setting and Checking File Permissions

The following example sets the file permission bits for a file named **/home/cnd/mod1**, then calls the *stat*( ) function to verify the permissions.

```
#include <sys/types.h>
#include <sys/stat.h>

int status;
struct stat buffer
...
chmod("home/cnd/mod1", S_IRWXU|S_IRWXG|S_IROTH|S_IWOTH);
status = stat("home/cnd/mod1", &buffer;);
```

APPLICATION USAGE

In order to ensure that the S_ISUID and S_ISGID bits are set, an application requiring this should use *stat*( ) after a successful *chmod*( ) to verify this.

Any file descriptors currently open by any process on the file could possibly become invalid if the mode of the file is changed to a value which would deny access to that process. One

situation where this could occur is on a stateless file system. This behavior will not occur in a conforming environment.

RATIONALE

This volume of IEEE Std 1003.1-200x specifies that the S_ISGID bit is cleared by *chmod*() on a regular file under certain conditions. This is specified on the assumption that regular files may be executed, and the system should prevent users from making executable *setgid*() files perform with privileges that the caller does not have. On implementations that support execution of other file types, the S_ISGID bit should be cleared for those file types under the same circumstances.

Implementations that use the S_ISUID bit to indicate some other function (for example, mandatory record locking) on non-executable files need not clear this bit on writing. They should clear the bit for executable files and any other cases where the bit grants special powers to processes that change the file contents. Similar comments apply to the S_ISGID bit.

FUTURE DIRECTIONS

None.

SEE ALSO

chown(), *mkdir*(), *mkfifo*(), *open*(), *stat*(), *statvfs*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [EINVAL] and [EINTR] optional error conditions are added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

6387 **NAME**

6388 chown — change owner and group of a file

6389 **SYNOPSIS**

6390 `#include <unistd.h>`

6391 `int chown(const char *path, uid_t owner, gid_t group);`

6392 **DESCRIPTION**

6393 The *chown*( ) function shall change the user and group ownership of a file.

6394 The *path* argument points to a pathname naming a file. The user ID and group ID of the named
6395 file shall be set to the numeric values contained in *owner* and *group*, respectively.

6396 Only processes with an effective user ID equal to the user ID of the file or with appropriate
6397 privileges may change the ownership of a file. If _POSIX_CHOWN_RESTRICTED is in effect for
6398 *path*:

6399 - Changing the user ID is restricted to processes with appropriate privileges.

6400 - Changing the group ID is permitted to a process with an effective user ID equal to the user
6401 ID of the file, but without appropriate privileges, if and only if *owner* is equal to the file's user
6402 ID or (**uid_t**)–1 and *group* is equal either to the calling process' effective group ID or to one of
6403 its supplementary group IDs.

6404 If the specified file is a regular file, one or more of the S_IXUSR, S_IXGRP, or S_IXOTH bits of
6405 the file mode are set, and the process does not have appropriate privileges, the set-user-ID
6406 (S_ISUID) and set-group-ID (S_ISGID) bits of the file mode shall be cleared upon successful
6407 return from *chown*( ). If the specified file is a regular file, one or more of the S_IXUSR, S_IXGRP,
6408 or S_IXOTH bits of the file mode are set, and the process has appropriate privileges, it is
6409 implementation-defined whether the set-user-ID and set-group-ID bits are altered. If the *chown*( )
6410 function is successfully invoked on a file that is not a regular file and one or more of the
6411 S_IXUSR, S_IXGRP, or S_IXOTH bits of the file mode are set, the set-user-ID and set-group-ID
6412 bits may be cleared.

6413 If *owner* or *group* is specified as (**uid_t**)–1 or (**gid_t**)–1, respectively, the corresponding ID of the
6414 file shall not be changed. If both owner and group are –1, the times need not be updated.

6415 Upon successful completion, *chown*( ) shall mark for update the *st_ctime* field of the file.

6416 **RETURN VALUE**

6417 Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to
6418 indicate the error. If –1 is returned, no changes are made in the user ID and group ID of the file.

6419 **ERRORS**

6420 The *chown*( ) function shall fail if:

6421 [EACCES] Search permission is denied on a component of the path prefix.

6422 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
6423 argument.

6424 [ENAMETOOLONG]
6425 The length of the *path* argument exceeds {PATH_MAX} or a pathname
6426 component is longer than {NAME_MAX}.

6427 [ENOTDIR] A component of the path prefix is not a directory.

6428 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

[EPERM] The effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges and _POSIX_CHOWN_RESTRICTED indicates that such privilege is required.

[EROFS] The named file resides on a read-only file system.

The *chown*( ) function may fail if:

[EIO] An I/O error occurred while reading or writing to the file system.

[EINTR] The *chown*( ) function was interrupted by a signal which was caught.

[EINVAL] The owner or group ID supplied is not a value supported by the implementation.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}. |

EXAMPLES

None.

APPLICATION USAGE

Although *chown*( ) can be used on some implementations by the file owner to change the owner and group to any desired values, the only portable use of this function is to change the group of a file to the effective GID of the calling process or to a member of its group set. |

RATIONALE

System III and System V allow a user to give away files; that is, the owner of a file may change its user ID to anything. This is a serious problem for implementations that are intended to meet government security regulations. Version 7 and 4.3 BSD permit only the superuser to change the user ID of a file. Some government agencies (usually not ones concerned directly with security) find this limitation too confining. This volume of IEEE Std 1003.1-200x uses *may* to permit secure implementations while not disallowing System V.

System III and System V allow the owner of a file to change the group ID to anything. Version 7 permits only the superuser to change the group ID of a file. 4.3 BSD permits the owner to change the group ID of a file to its effective group ID or to any of the groups in the list of supplementary group IDs, but to no others.

The POSIX.1-1990 standard requires that the *chown*( ) function invoked by a non-appropriate privileged process clear the S_ISGID and the S_ISUID bits for regular files, and permits them to be cleared for other types of files. This is so that changes in accessibility do not accidentally cause files to become security holes. Unfortunately, requiring these bits to be cleared on non-executable data files also clears the mandatory file locking bit (shared with S_ISGID), which is an extension on many implementations (it first appeared in System V). These bits should only be required to be cleared on regular files that have one or more of their execute bits set.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *pathconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The wording describing the optional dependency on _POSIX_CHOWN_RESTRICTED is restored.
- The [EPERM] error is restored as an error dependent on _POSIX_CHOWN_RESTRICTED. This is since its operand is a pathname and applications should be aware that the error may not occur for that pathname if the file system does not support _POSIX_CHOWN_RESTRICTED.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The value for *owner* of (**uid_t**)–1 allows the use of –1 by the owner of a file to change the group ID only. A corresponding change is made for group.
- The [ELOOP] mandatory error condition is added.
- The [EIO] and [EINTR] optional error conditions are added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Clarification is added that the S_ISUID and S_ISGID bits do not need to be cleared when the process has appropriate privileges.
- The [ELOOP] optional error condition is added.

NAME

cimag, cimagf, cimagl — complex imaginary functions

SYNOPSIS

```
#include <complex.h>

double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the imaginary part of *z*.

RETURN VALUE

These functions shall return the imaginary part value (as a real).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

For a variable *z* of complex type:

```
z == creal(z) + cimag(z)*I
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

carg( ), *conj*( ), *cproj*( ), *creal*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

6525 **NAME**

6526 clearerr — clear indicators on a stream

6527 **SYNOPSIS**

6528 `#include <stdio.h>`

6529 `void clearerr(FILE *stream);`

6530 **DESCRIPTION**

6531 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6532 conflict between the requirements described here and the ISO C standard is unintentional. This
6533 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

6534 The *clearerr*( ) function shall clear the end-of-file and error indicators for the stream to which
6535 *stream* points.

6536 **RETURN VALUE**

6537 The *clearerr*( ) function shall not return a value.

6538 **ERRORS**

6539 No errors are defined.

6540 **EXAMPLES**

6541 None.

6542 **APPLICATION USAGE**

6543 None.

6544 **RATIONALE**

6545 None.

6546 **FUTURE DIRECTIONS**

6547 None.

6548 **SEE ALSO**

6549 The Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

6550 **CHANGE HISTORY**

6551 First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

clock — report CPU time used

SYNOPSIS

```
#include <time.h>

clock_t clock(void);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *clock*( ) function shall return the implementation's best approximation to the processor time used by the process since the beginning of an implementation-defined era related only to the process invocation.

RETURN VALUE

To determine the time in seconds, the value returned by *clock*( ) should be divided by the value XSI of the macro CLOCKS_PER_SEC. CLOCKS_PER_SEC is defined to be one million in **<time.h>**. | If the processor time used is not available or its value cannot be represented, the function shall return the value (**clock_t**)–1.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

In order to measure the time spent in a program, *clock*( ) should be called at the start of the program and its return value subtracted from the value returned by subsequent calls. The value returned by *clock*( ) is defined for compatibility across systems that have clocks with different resolutions. The resolution on any particular system need not be to microsecond accuracy.

The value returned by *clock*( ) may wrap around on some implementations. For example, on a | machine with 32-bit values for **clock_t**, it wraps after 2 147 seconds or 36 minutes. |

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime( ), *ctime*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

6589 **NAME**

6590 clock_getcpuclockid — access a process CPU-time clock (**ADVANCED REALTIME**)

6591 **SYNOPSIS**

6592 CPT
```
#include <time.h>
```

6593
```
int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);
```

6594

6595 **DESCRIPTION**

6596 The *clock_getcpuclockid*( ) function shall return the clock ID of the CPU-time clock of the process
6597 specified by *pid*. If the process described by *pid* exists and the calling process has permission,
6598 the clock ID of this clock shall be returned in *clock_id*.

6599 If *pid* is zero, the *clock_getcpuclockid*( ) function shall return the clock ID of the CPU-time clock of
6600 the process making the call, in *clock_id*.

6601 The conditions under which one process has permission to obtain the CPU-time clock ID of
6602 other processes are implementation-defined.

6603 **RETURN VALUE**

6604 Upon successful completion, *clock_getcpuclockid*( ) shall return zero; otherwise, an error number
6605 shall be returned to indicate the error.

6606 **ERRORS**

6607 The *clock_getcpuclockid*( ) function shall fail if:

6608 [EPERM] The requesting process does not have permission to access the CPU-time
6609 clock for the process.

6610 The *clock_getcpuclockid*( ) function may fail if:

6611 [ESRCH] No process can be found corresponding to the process specified by *pid*.

6612 **EXAMPLES**

6613 None.

6614 **APPLICATION USAGE**

6615 The *clock_getcpuclockid*( ) function is part of the Process CPU-Time Clocks option and need not
6616 be provided on all implementations.

6617 **RATIONALE**

6618 None.

6619 **FUTURE DIRECTIONS**

6620 None.

6621 **SEE ALSO**

6622 *clock_getres*( ), *timer_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

6623 **CHANGE HISTORY**

6624 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

6625 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

clock_getres, clock_gettime, clock_settime — clock and timer functions (**REALTIME**)

SYNOPSIS

TMR
```
#include <time.h>

int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

DESCRIPTION

The *clock_getres*( ) function shall return the resolution of any clock. Clock resolutions are implementation-defined and cannot be set by a process. If the argument *res* is not NULL, the resolution of the specified clock shall be stored in the location pointed to by *res*. If *res* is NULL, the clock resolution is not returned. If the *time* argument of *clock_settime*( ) is not a multiple of *res*, then the value is truncated to a multiple of *res*.

The *clock_gettime*( ) function shall return the current value *tp* for the specified clock, *clock_id*.

The *clock_settime*( ) function shall set the specified clock, *clock_id*, to the value specified by *tp*. Time values that are between two consecutive non-negative integer multiples of the resolution of the specified clock shall be truncated down to the smaller multiple of the resolution.

A clock may be system-wide (that is, visible to all processes) or per-process (measuring time that is meaningful only within a process). All implementations shall support a *clock_id* of CLOCK_REALTIME as defined in **<time.h>**. This clock represents the realtime clock for the system. For this clock, the values returned by *clock_gettime*( ) and specified by *clock_settime*( ) represent the amount of time (in seconds and nanoseconds) since the Epoch. An implementation may also support additional clocks. The interpretation of time values for these clocks is unspecified.

If the value of the CLOCK_REALTIME clock is set via *clock_settime*( ), the new value of the clock shall be used to determine the time of expiration for absolute time services based upon the CLOCK_REALTIME clock. This applies to the time at which armed absolute timers expire. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the time service shall expire immediately as if the clock had reached the requested time normally.

Setting the value of the CLOCK_REALTIME clock via *clock_settime*( ) shall have no effect on threads that are blocked waiting for a relative time service based upon this clock, including the *nanosleep*( ) function; nor on the expiration of relative timers based upon this clock. Consequently, these time services shall expire when the requested relative interval elapses, independently of the new or old value of the clock.

MON If the Monotonic Clock option is supported, all implementations shall support a *clock_id* of CLOCK_MONOTONIC defined in **<time.h>**. This clock represents the monotonic clock for the system. For this clock, the value returned by *clock_gettime*( ) represents the amount of time (in seconds and nanoseconds) since an unspecified point in the past (for example, system start-up time, or the Epoch). This point does not change after system start-up time. The value of the CLOCK_MONOTONIC clock cannot be set via *clock_settime*( ). This function shall fail if it is invoked with a *clock_id* argument of CLOCK_MONOTONIC.

The effect of setting a clock via *clock_settime*( ) on armed per-process timers associated with a clock other than CLOCK_REALTIME is implementation-defined.

CS If the value of the CLOCK_REALTIME clock is set via *clock_settime*( ), the new value of the clock shall be used to determine the time at which the system shall awaken a thread blocked on an

absolute *clock_nanosleep*( ) call based upon the CLOCK_REALTIME clock. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the call shall return immediately as if the clock had reached the requested time normally.

Setting the value of the CLOCK_REALTIME clock via *clock_settime*( ) shall have no effect on any thread that is blocked on a relative *clock_nanosleep*( ) call. Consequently, the call shall return when the requested relative interval elapses, independently of the new or old value of the clock.

The appropriate privilege to set a particular clock is implementation-defined.

CPT If _POSIX_CPUTIME is defined, implementations shall support clock ID values obtained by invoking *clock_getcpuclockid*( ), which represent the CPU-time clock of a given process. Implementations shall also support the special **clockid_t** value CLOCK_PROCESS_CPUTIME_ID, which represents the CPU-time clock of the calling process when invoking one of the *clock_**( ) or *timer_**( ) functions. For these clock IDs, the values returned by *clock_gettime*( ) and specified by *clock_settime*( ) represent the amount of execution time of the process associated with the clock. Changing the value of a CPU-time clock via *clock_settime*( ) shall have no effect on the behavior of the sporadic server scheduling policy (see **Scheduling Policies** (on page 494)).

TCT If _POSIX_THREAD_CPUTIME is defined, implementations shall support clock ID values obtained by invoking *pthread_getcpuclockid*( ), which represent the CPU-time clock of a given thread. Implementations shall also support the special **clockid_t** value CLOCK_THREAD_CPUTIME_ID, which represents the CPU-time clock of the calling thread when invoking one of the *clock_**( ) or *timer_**( ) functions. For these clock IDs, the values returned by *clock_gettime*( ) and specified by *clock_settime*( ) shall represent the amount of execution time of the thread associated with the clock. Changing the value of a CPU-time clock via *clock_settime*( ) shall have no effect on the behavior of the sporadic server scheduling policy (see **Scheduling Policies** (on page 494)).

RETURN VALUE

A return value of 0 shall indicate that the call succeeded. A return value of –1 shall indicate that an error occurred, and *errno* shall be set to indicate the error.

ERRORS

The *clock_getres*( ), *clock_gettime*( ), and *clock_settime*( ) functions shall fail if:

[EINVAL] The *clock_id* argument does not specify a known clock.

The *clock_settime*( ) function shall fail if:

[EINVAL] The *tp* argument to *clock_settime*( ) is outside the range for the given clock ID.

[EINVAL] The *tp* argument specified a nanosecond value less than zero or greater than or equal to 1 000 million.

MON [EINVAL] The value of the *clock_id* argument is CLOCK_MONOTONIC.

The *clock_settime*( ) function may fail if:

[EPERM] The requesting process does not have the appropriate privilege to set the specified clock.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Timers option and need not be available on all implementations.

Note that the absolute value of the monotonic clock is meaningless (because its origin is arbitrary), and thus there is no need to set it. Furthermore, realtime applications can rely on the fact that the value of this clock is never set and, therefore, that time intervals measured with this clock will not be affected by calls to *clock_settime*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getcpuclockid( ), *clock_nanosleep*( ), *ctime*( ), *mq_timedreceive*( ), *mq_timedsend*( ), *nanosleep*( ), *pthread_mutex_timedlock*( ), *sem_timedwait*( ), *time*( ), *timer_create*( ), *timer_getoverrun*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

The APPLICATION USAGE section is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Clarification is added of the effect of resetting the clock resolution.

CPU-time clocks and the *clock_getcpuclockid*( ) function are added for alignment with IEEE Std 1003.1d-1999.

The following changes are added for alignment with IEEE Std 1003.1j-2000:

- The DESCRIPTION is updated as follows:
  - The value returned by *clock_gettime*( ) for CLOCK_MONOTONIC is specified.
  - *clock_settime*( ) failing for CLOCK_MONOTONIC is specified.
  - The effects of *clock_settime*( ) on the *clock_nanosleep*( ) function with respect to CLOCK_REALTIME is specified.
- An [EINVAL] error is added to the ERRORS section, indicating that *clock_settime*( ) fails for CLOCK_MONOTONIC.
- The APPLICATION USAGE section notes that the CLOCK_MONOTONIC clock need not and shall not be set by *clock_settime*( ) since the absolute value of the CLOCK_MONOTONIC clock is meaningless.
- The *clock_nanosleep*( ), *mq_timedreceive*( ), *mq_timedsend*( ), *pthread_mutex_timedlock*( ), *sem_timedwait*( ), *timer_create*( ), and *timer_settime*( ) functions are added to the SEE ALSO section.

6752 **NAME**

6753 clock_nanosleep — high resolution sleep with specifiable clock (**ADVANCED REALTIME**)

6754 **SYNOPSIS**

6755 CS
```
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
    const struct timespec *rqtp, struct timespec *rmtp);
```
6758

6759 **DESCRIPTION**

6760 If the flag TIMER_ABSTIME is not set in the *flags* argument, the *clock_nanosleep*( ) function shall
6761 cause the current thread to be suspended from execution until either the time interval specified
6762 by the *rqtp* argument has elapsed, or a signal is delivered to the calling thread and its action is to
6763 invoke a signal-catching function, or the process is terminated. The clock used to measure the
6764 time shall be the clock specified by *clock_id*.

6765 If the flag TIMER_ABSTIME is set in the *flags* argument, the *clock_nanosleep*( ) function shall
6766 cause the current thread to be suspended from execution until either the time value of the clock
6767 specified by *clock_id* reaches the absolute time specified by the *rqtp* argument, or a signal is
6768 delivered to the calling thread and its action is to invoke a signal-catching function, or the
6769 process is terminated. If, at the time of the call, the time value specified by *rqtp* is less than or
6770 equal to the time value of the specified clock, then *clock_nanosleep*( ) shall return immediately
6771 and the calling process shall not be suspended.

6772 The suspension time caused by this function may be longer than requested because the
6773 argument value is rounded up to an integer multiple of the sleep resolution, or because of the
6774 scheduling of other activity by the system. But, except for the case of being interrupted by a
6775 signal, the suspension time for the relative *clock_nanosleep*( ) function (that is, with the
6776 TIMER_ABSTIME flag not set) shall not be less than the time interval specified by *rqtp*, as
6777 measured by the corresponding clock. The suspension for the absolute *clock_nanosleep*( ) function
6778 (that is, with the TIMER_ABSTIME flag set) shall be in effect at least until the value of the
6779 corresponding clock reaches the absolute time specified by *rqtp*, except for the case of being
6780 interrupted by a signal.

6781 The use of the *clock_nanosleep*( ) function shall have no effect on the action or blockage of any
6782 signal.

6783 The *clock_nanosleep*( ) function shall fail if the *clock_id* argument refers to the CPU-time clock of
6784 the calling thread. It is unspecified if *clock_id* values of other CPU-time clocks are allowed.

6785 **RETURN VALUE**

6786 If the *clock_nanosleep*( ) function returns because the requested time has elapsed, its return value
6787 shall be zero.

6788 If the *clock_nanosleep*( ) function returns because it has been interrupted by a signal, it shall return
6789 the corresponding error value. For the relative *clock_nanosleep*( ) function, if the *rmtp* argument is
6790 non-NULL, the **timespec** structure referenced by it shall be updated to contain the amount of
6791 time remaining in the interval (the requested time minus the time actually slept). If the *rmtp*
6792 argument is NULL, the remaining time is not returned. The absolute *clock_nanosleep*( ) function
6793 has no effect on the structure referenced by *rmtp*.

6794 If *clock_nanosleep*( ) fails, it shall return the corresponding error value.

6795 **ERRORS**

6796 The *clock_nanosleep*( ) function shall fail if:

6797 [EINTR] The *clock_nanosleep*( ) function was interrupted by a signal.

6798 [EINVAL] The *rqtp* argument specified a nanosecond value less than zero or greater than
6799 or equal to 1 000 million; or the TIMER_ABSTIME flag was specified in flags
6800 and the *rqtp* argument is outside the range for the clock specified by *clock_id*;
6801 or the *clock_id* argument does not specify a known clock, or specifies the
6802 CPU-time clock of the calling thread.

6803 [ENOTSUP] The *clock_id* argument specifies a clock for which *clock_nanosleep*( ) is not
6804 supported, such as a CPU-time clock.

6805 **EXAMPLES**

6806 None.

6807 **APPLICATION USAGE**

6808 Calling *clock_nanosleep*( ) with the value TIMER_ABSTIME not set in the *flags* argument and with
6809 a *clock_id* of CLOCK_REALTIME is equivalent to calling *nanosleep*( ) with the same *rqtp* and *rmtp*
6810 arguments.

6811 **RATIONALE**

6812 The *nanosleep*( ) function specifies that the system-wide clock CLOCK_REALTIME is used to
6813 measure the elapsed time for this time service. However, with the introduction of the monotonic
6814 clock CLOCK_MONOTONIC a new relative sleep function is needed to allow an application to
6815 take advantage of the special characteristics of this clock.

6816 There are many applications in which a process needs to be suspended and then activated
6817 multiple times in a periodic way; for example, to poll the status of a non-interrupting device or
6818 to refresh a display device. For these cases, it is known that precise periodic activation cannot be
6819 achieved with a relative *sleep*( ) or *nanosleep*( ) function call. Suppose, for example, a periodic
6820 process that is activated at time $T0$, executes for a while, and then wants to suspend itself until
6821 time $T0+T$, the period being T. If this process wants to use the *nanosleep*( ) function, it must first
6822 call *clock_gettime*( ) to get the current time, then calculate the difference between the current time
6823 and $T0+T$ and, finally, call *nanosleep*( ) using the computed interval. However, the process could
6824 be preempted by a different process between the two function calls, and in this case the interval
6825 computed would be wrong; the process would wake up later than desired. This problem would
6826 not occur with the absolute *clock_nanosleep*( ) function, since only one function call would be
6827 necessary to suspend the process until the desired time. In other cases, however, a relative sleep
6828 is needed, and that is why both functionalities are required.

6829 Although it is possible to implement periodic processes using the timers interface, this
6830 implementation would require the use of signals, and the reservation of some signal numbers. In
6831 this regard, the reasons for including an absolute version of the *clock_nanosleep*( ) function in
6832 IEEE Std 1003.1-200x are the same as for the inclusion of the relative *nanosleep*( ).

6833 It is also possible to implement precise periodic processes using *pthread_cond_timedwait*( ), in
6834 which an absolute timeout is specified that takes effect if the condition variable involved is
6835 never signaled. However, the use of this interface is unnatural, and involves performing other
6836 operations on mutexes and condition variables that imply an unnecessary overhead.
6837 Furthermore, *pthread_cond_timedwait*( ) is not available in implementations that do not support
6838 threads.

6839 Although the interface of the relative and absolute versions of the new high resolution sleep
6840 service is the same *clock_nanosleep*( ) function, the *rmtp* argument is only used in the relative
6841 sleep. This argument is needed in the relative *clock_nanosleep*( ) function to reissue the function

call if it is interrupted by a signal, but it is not needed in the absolute *clock_nanosleep*( ) function call; if the call is interrupted by a signal, the absolute *clock_nanosleep*( ) function can be invoked again with the same *rqtp* argument used in the interrupted call.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getres( ), *nanosleep*( ), *pthread_cond_timedwait*( ), *sleep*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

6852 **NAME**

6853 clock_settime — clock and timer functions (**REALTIME**)

6854 **SYNOPSIS**

6855 TMR
```
#include <time.h>
```

6856
```
int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

6857

6858 **DESCRIPTION**

6859 Refer to *clock_getres*( ).

6860 **NAME**

6861 clog, clogf, clogl — complex natural logarithm functions

6862 **SYNOPSIS**

6863
```
#include <complex.h>
```

6864
```
double complex clog(double complex z);
```
6865
```
float complex clogf(float complex z);
```
6866
```
long double complex clogl(long double complex z);
```

6867 **DESCRIPTION**

6868 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6869 conflict between the requirements described here and the ISO C standard is unintentional. This
6870 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

6871 These functions shall compute the complex natural (base *e*) logarithm of *z*, with a branch cut
6872 along the negative real axis.

6873 **RETURN VALUE**

6874 These functions shall return the complex natural logarithm value, in the range of a strip
6875 mathematically unbounded along the real axis and in the interval [$-i\pi$, $+i\pi$] along the imaginary
6876 axis.

6877 **ERRORS**

6878 No errors are defined.

6879 **EXAMPLES**

6880 None.

6881 **APPLICATION USAGE**

6882 None.

6883 **RATIONALE**

6884 None.

6885 **FUTURE DIRECTIONS**

6886 None.

6887 **SEE ALSO**

6888 *cexp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

6889 **CHANGE HISTORY**

6890 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

close — close a file descriptor

SYNOPSIS

```
#include <unistd.h>

int close(int fildes);
```

DESCRIPTION

The *close*( ) function shall deallocate the file descriptor indicated by *fildes*. To deallocate means to make the file descriptor available for return by subsequent calls to *open*( ) or other functions that allocate file descriptors. All outstanding record locks owned by the process on the file associated with the file descriptor shall be removed (that is, unlocked).

If *close*( ) is interrupted by a signal that is to be caught, it shall return –1 with *errno* set to [EINTR] and the state of *fildes* is unspecified. If an I/O error occurred while reading from or writing to the file system during *close*( ), it may return –1 with *errno* set to [EIO]; if this error is returned, the state of *fildes* is unspecified.

When all file descriptors associated with a pipe or FIFO special file are closed, any data remaining in the pipe or FIFO shall be discarded.

When all file descriptors associated with an open file description have been closed the open file description shall be freed.

If the link count of the file is 0, when all file descriptors associated with the file are closed, the space occupied by the file shall be freed and the file shall no longer be accessible.

XSR If a STREAMS-based *fildes* is closed and the calling process was previously registered to receive a SIGPOLL signal for events associated with that STREAM, the calling process shall be unregistered for events associated with the STREAM. The last *close*( ) for a STREAM shall cause the STREAM associated with *fildes* to be dismantled. If O_NONBLOCK is not set and there have been no signals posted for the STREAM, and if there is data on the module's write queue, *close*( ) shall wait for an unspecified time (for each module and driver) for any output to drain before dismantling the STREAM. The time delay can be changed via an I_SETCLTIME *ioctl*( ) request. If the O_NONBLOCK flag is set, or if there are any pending signals, *close*( ) shall not wait for output to drain, and shall dismantle the STREAM immediately.

If the implementation supports STREAMS-based pipes, and *fildes* is associated with one end of a pipe, the last *close*( ) shall cause a hangup to occur on the other end of the pipe. In addition, if the other end of the pipe has been named by *fattach*( ), then the last *close*( ) shall force the named end to be detached by *fdetach*( ). If the named end has no open file descriptors associated with it and gets detached, the STREAM associated with that end shall also be dismantled.

XSI If *fildes* refers to the master side of a pseudo-terminal, and this is the last close, a SIGHUP signal shall be sent to the process group, if any, for which the slave side of the pseudo-terminal is the controlling terminal. It is unspecified whether closing the master side of the pseudo-terminal flushes all queued input and output.

XSR If *fildes* refers to the slave side of a STREAMS-based pseudo-terminal, a zero-length message may be sent to the master.

AIO When there is an outstanding cancelable asynchronous I/O operation against *fildes* when *close*( ) is called, that I/O operation may be canceled. An I/O operation that is not canceled completes as if the *close*( ) operation had not yet occurred. All operations that are not canceled shall complete as if the *close*( ) blocked until the operations completed. The *close*( ) operation itself need not block awaiting such I/O completion. Whether any I/O operation is canceled, and which I/O operation may be canceled upon *close*( ), is implementation-defined.

MF|SHM If a shared memory object or a memory mapped file remains referenced at the last close (that is, a process has it mapped), then the entire contents of the memory object shall persist until the memory object becomes unreferenced. If this is the last close of a shared memory object or a memory mapped file and the close results in the memory object becoming unreferenced, and the memory object has been unlinked, then the memory object shall be removed.

If *fildes* refers to a socket, *close*( ) shall cause the socket to be destroyed. If the socket is in connection-mode, and the SO_LINGER option is set for the socket with non-zero linger time, and the socket has untransmitted data, then *close*( ) shall block for up to the current linger interval until all data is transmitted.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *close*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINTR] The *close*( ) function was interrupted by a signal.

The *close*( ) function may fail if:

[EIO] An I/O error occurred while reading from or writing to the file system.

EXAMPLES

Reassigning a File Descriptor

The following example closes the file descriptor associated with standard output for the current process, re-assigns standard output to a new file descriptor, and closes the original file descriptor to clean up. This example assumes that the file descriptor 0 (which is the descriptor for standard input) is not closed.

```
#include <unistd.h>
...
int pfd;
...
close(1);
dup(pfd);
close(pfd);
...
```

Incidentally, this is exactly what could be achieved using:

```
dup2(pfd, 1);
close(pfd);
```

Closing a File Descriptor

In the following example, *close*( ) is used to close a file descriptor after an unsuccessful attempt is made to associate that file descriptor with a stream.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

```
#define LOCKFILE "/etc/ptmp"
...
int pfd;
FILE *fpfd;
...
if ((fpfd = fdopen (pfd, "w")) == NULL) {
    close(pfd);
    unlink(LOCKFILE);
    exit(1);
}
...
```

APPLICATION USAGE

An application that had used the *stdio* routine *fopen*( ) to open a file should use the corresponding *fclose*( ) routine rather than *close*( ). Once a file is closed, the file descriptor no longer exists, since the integer corresponding to it no longer refers to a file.

RATIONALE

The use of interruptible device close routines should be discouraged to avoid problems with the implicit closes of file descriptors by *exec* and *exit*( ). This volume of IEEE Std 1003.1-200x only intends to permit such behavior by specifying the [EINTR] error condition.

FUTURE DIRECTIONS

None.

SEE ALSO

fattach( ), *fclose*( ), *fdetach*( ), *fopen*( ), *ioctl*( ), *open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**, Section 2.6 (on page 488)

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

Issue 6

The DESCRIPTION related to a STREAMS-based file or pseudo-terminal is marked as part of the XSI STREAMS Option Group.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] error condition is added as an optional error.
- The DESCRIPTION is updated to describe the state of the *fildes* file descriptor as unspecified if an I/O error occurs and an [EIO] error condition is returned.

Text referring to sockets is added to the DESCRIPTION.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that shared memory objects and memory mapped files (and not typed memory objects) are the types of memory objects to which the paragraph on last closes applies.

7018 **NAME**

7019 closedir — close a directory stream

7020 **SYNOPSIS**

7021 ```
#include <dirent.h>
```

7022 ```
int closedir(DIR *dirp);
```

7023 **DESCRIPTION**

7024 The *closedir*( ) function shall close the directory stream referred to by the argument *dirp*. Upon
7025 return, the value of *dirp* may no longer point to an accessible object of the type **DIR**. If a file
7026 descriptor is used to implement type **DIR**, that file descriptor shall be closed.

7027 **RETURN VALUE**

7028 Upon successful completion, *closedir*( ) shall return 0; otherwise, –1 shall be returned and *errno*
7029 set to indicate the error.

7030 **ERRORS**

7031 The *closedir*( ) function may fail if:

7032 [EBADF] The *dirp* argument does not refer to an open directory stream.

7033 [EINTR] The *closedir*( ) function was interrupted by a signal.

7034 **EXAMPLES**

7035 **Closing a Directory Stream**

7036 The following program fragment demonstrates how the *closedir*( ) function is used.

```
7037 ...
7038     DIR *dir;
7039     struct dirent *dp;
7040 ...
7041     if ((dir = opendir (".")) == NULL) {
7042 ...
7043     }
7044     while ((dp = readdir (dir)) != NULL) {
7045 ...
7046     }
7047     closedir(dir);
7048 ...
```

7049 **APPLICATION USAGE**

7050 None.

7051 **RATIONALE**

7052 None.

7053 **FUTURE DIRECTIONS**

7054 None.

7055 **SEE ALSO**

7056 *opendir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [EINTR] error condition is added as an optional error condition.

7067 **NAME**

7068 closelog, openlog, setlogmask, syslog — control system log

7069 **SYNOPSIS**

7070 XSI
```
#include <syslog.h>

void closelog(void);
void openlog(const char *ident, int logopt, int facility);
int setlogmask(int maskpri);
void syslog(int priority, const char *message, ... /* arguments */);
```
7071
7072
7073
7074
7075

7076 **DESCRIPTION**

7077 The *syslog*( ) function shall send a message to an implementation-defined logging facility, which
7078 may log it in an implementation-defined system log, write it to the system console, forward it to
7079 a list of users, or forward it to the logging facility on another host over the network. The logged
7080 message shall include a message header and a message body. The message header contains at
7081 least a timestamp and a tag string.

7082 The message body is generated from the *message* and following arguments in the same manner
7083 as if these were arguments to *printf*( ), except that the additional conversion specification `%m`
7084 shall be recognized; it shall convert no arguments, shall cause the output of the error message
7085 string associated with the value of *errno* on entry to *syslog*( ), and may be mixed with argument
7086 specifications of the `"%`*n*`$"` form. If a complete conversion specification with the `m` conversion
7087 specifier character is not just `%m`, the behavior is undefined. A trailing <newline> may be added
7088 if needed.

7089 Values of the *priority* argument are formed by OR'ing together a severity level value and an
7090 optional facility value. If no facility value is specified, the current default facility value is used.

7091 Possible values of severity level include:

7092 LOG_EMERG A panic condition.

7093 LOG_ALERT A condition that should be corrected immediately, such as a corrupted system
7094 database.

7095 LOG_CRIT Critical conditions, such as hard device errors.

7096 LOG_ERR Errors.

7097 LOG_WARNING
7098 Warning messages.

7099 LOG_NOTICE Conditions that are not error conditions, but that may require special
7100 handling.

7101 LOG_INFO Informational messages.

7102 LOG_DEBUG Messages that contain information normally of use only when debugging a
7103 program.

7104 The facility indicates the application or system component generating the message. Possible
7105 facility values include:

7106 LOG_USER Messages generated by arbitrary processes. This is the default facility
7107 identifier if none is specified.

7108 LOG_LOCAL0 Reserved for local use.

LOG_LOCAL1	Reserved for local use.
LOG_LOCAL2	Reserved for local use.
LOG_LOCAL3	Reserved for local use.
LOG_LOCAL4	Reserved for local use.
LOG_LOCAL5	Reserved for local use.
LOG_LOCAL6	Reserved for local use.
LOG_LOCAL7	Reserved for local use.

The *openlog*( ) function shall set process attributes that affect subsequent calls to *syslog*( ). The *ident* argument is a string that is prepended to every message. The *logopt* argument indicates logging options. Values for *logopt* are constructed by a bitwise-inclusive OR of zero or more of the following:

LOG_PID	Log the process ID with each message. This is useful for identifying specific processes.
LOG_CONS	Write messages to the system console if they cannot be sent to the logging facility. The *syslog*( ) function ensures that the process does not acquire the console as a controlling terminal in the process of writing the message.
LOG_NDELAY	Open the connection to the logging facility immediately. Normally the open is delayed until the first message is logged. This is useful for programs that need to manage the order in which file descriptors are allocated.
LOG_ODELAY	Delay open until *syslog*( ) is called.
LOG_NOWAIT	Do not wait for child processes that may have been created during the course of logging the message. This option should be used by processes that enable notification of child termination using SIGCHLD, since *syslog*( ) may otherwise block waiting for a child whose exit status has already been collected.

The *facility* argument encodes a default facility to be assigned to all messages that do not have an explicit facility already encoded. The initial default facility is LOG_USER.

The *openlog*( ) and *syslog*( ) functions may allocate a file descriptor. It is not necessary to call *openlog*( ) prior to calling *syslog*( ).

The *closelog*( ) function shall close any open file descriptors allocated by previous calls to *openlog*( ) or *syslog*( ).

The *setlogmask*( ) function shall set the log priority mask for the current process to *maskpri* and return the previous mask. If the *maskpri* argument is 0, the current log mask is not modified. Calls by the current process to *syslog*( ) with a priority not set in *maskpri* shall be rejected. The default log mask allows all priorities to be logged. A call to *openlog*( ) is not required prior to calling *setlogmask*( ).

Symbolic constants for use as values of the *logopt*, *facility*, *priority*, and *maskpri* arguments are defined in the **<syslog.h>** header.

RETURN VALUE

The *setlogmask*( ) function shall return the previous log priority mask. The *closelog*( ), *openlog*( ), and *syslog*( ) functions shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

Using openlog()

The following example causes subsequent calls to *syslog*( ) to log the process ID with each message, and to write messages to the system console if they cannot be sent to the logging facility.

```
#include <syslog.h>

char *ident = "Process demo";
int logopt = LOG_PID | LOG_CONS;
int facility = LOG_USER;
...
openlog(ident, logopt, facility);
```

Using setlogmask()

The following example causes subsequent calls to *syslog*( ) to accept error messages or messages generated by arbitrary processes, and to reject all other messages.

```
#include <syslog.h>

int result;
int mask = LOG_MASK (LOG_ERR | LOG_USER);
...
result = setlogmask(mask);
```

Using syslog

The following example sends the message `"This is a message"` to the default logging facility, marking the message as an error message generated by random processes.

```
#include <syslog.h>

char *message = "This is a message";
int priority = LOG_ERR | LOG_USER;
...
syslog(priority, message);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

printf( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<syslog.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

7191 **NAME**

7192 confstr — get configurable variables

7193 **SYNOPSIS**

7194 ```
#include <unistd.h>
```

7195 ```
size_t confstr(int name, char *buf, size_t len);
```

7196 **DESCRIPTION**

7197 The *confstr*( ) function shall return configuration-defined string values. Its use and purpose are
7198 similar to *sysconf*( ), but it is used where string values rather than numeric values are returned.

7199 The *name* argument represents the system variable to be queried. The implementation shall
7200 support the following name values, defined in **<unistd.h>**. It may support others:

7201 _CS_PATH
7202 _CS_POSIX_V6_ILP32_OFF32_CFLAGS
7203 _CS_POSIX_V6_ILP32_OFF32_LDFLAGS
7204 _CS_POSIX_V6_ILP32_OFF32_LIBS
7205 _CS_POSIX_V6_ILP32_OFF32_LINTFLAGS
7206 _CS_POSIX_V6_ILP32_OFFBIG_CFLAGS
7207 _CS_POSIX_V6_ILP32_OFFBIG_LDFLAGS
7208 _CS_POSIX_V6_ILP32_OFFBIG_LIBS
7209 _CS_POSIX_V6_ILP32_OFFBIG_LINTFLAGS
7210 _CS_POSIX_V6_LP64_OFF64_CFLAGS
7211 _CS_POSIX_V6_LP64_OFF64_LDFLAGS
7212 _CS_POSIX_V6_LP64_OFF64_LIBS
7213 _CS_POSIX_V6_LP64_OFF64_LINTFLAGS
7214 _CS_POSIX_V6_LPBIG_OFFBIG_CFLAGS
7215 _CS_POSIX_V6_LPBIG_OFFBIG_LDFLAGS
7216 _CS_POSIX_V6_LPBIG_OFFBIG_LIBS
7217 _CS_POSIX_V6_LPBIG_OFFBIG_LINTFLAGS
7218 XSI _CS_XBS5_ILP32_OFF32_CFLAGS (**LEGACY**)
7219 _CS_XBS5_ILP32_OFF32_LDFLAGS (**LEGACY**)
7220 _CS_XBS5_ILP32_OFF32_LIBS (**LEGACY**)
7221 _CS_XBS5_ILP32_OFF32_LINTFLAGS (**LEGACY**)
7222 _CS_XBS5_ILP32_OFFBIG_CFLAGS (**LEGACY**)
7223 _CS_XBS5_ILP32_OFFBIG_LDFLAGS (**LEGACY**)
7224 _CS_XBS5_ILP32_OFFBIG_LIBS (**LEGACY**)
7225 _CS_XBS5_ILP32_OFFBIG_LINTFLAGS (**LEGACY**)
7226 _CS_XBS5_LP64_OFF64_CFLAGS (**LEGACY**)
7227 _CS_XBS5_LP64_OFF64_LDFLAGS (**LEGACY**)
7228 _CS_XBS5_LP64_OFF64_LIBS (**LEGACY**)
7229 _CS_XBS5_LP64_OFF64_LINTFLAGS (**LEGACY**)
7230 _CS_XBS5_LPBIG_OFFBIG_CFLAGS (**LEGACY**)
7231 _CS_XBS5_LPBIG_OFFBIG_LDFLAGS (**LEGACY**)
7232 _CS_XBS5_LPBIG_OFFBIG_LIBS (**LEGACY**)
7233 _CS_XBS5_LPBIG_OFFBIG_LINTFLAGS (**LEGACY**)
7234

7235 If *len* is not 0, and if *name* has a configuration-defined value, *confstr*( ) shall copy that value into
7236 the *len*-byte buffer pointed to by *buf*. If the string to be returned is longer than *len* bytes,
7237 including the terminating null, then *confstr*( ) shall truncate the string to *len*–1 bytes and null-
7238 terminate the result. The application can detect that the string was truncated by comparing the
7239 value returned by *confstr*( ) with *len*.

If *len* is 0 and *buf* is a null pointer, then *confstr*( ) shall still return the integer value as defined below, but shall not return a string. If *len* is 0 but *buf* is not a null pointer, the result is unspecified.

If the implementation supports the Shell option, the string stored in *buf* after a call to:

```
confstr(_CS_PATH, buf, sizeof(buf))
```

can be used as a value of the *PATH* environment variable that accesses all of the standard utilities of IEEE Std 1003.1-200x, if the return value is less than or equal to *sizeof*(*buf*).

RETURN VALUE

If *name* has a configuration-defined value, *confstr*( ) shall return the size of buffer that would be needed to hold the entire configuration-defined value including the terminating null. If this return value is greater than *len*, the string returned in *buf* is truncated.

If *name* is invalid, *confstr*( ) shall return 0 and set *errno* to indicate the error.

If *name* does not have a configuration-defined value, *confstr*( ) shall return 0 and leave *errno* unchanged.

ERRORS

The *confstr*( ) function shall fail if:

[EINVAL] The value of the *name* argument is invalid.

EXAMPLES

None.

APPLICATION USAGE

An application can distinguish between an invalid *name* parameter value and one that corresponds to a configurable variable that has no configuration-defined value by checking if *errno* is modified. This mirrors the behavior of *sysconf*( ).

The original need for this function was to provide a way of finding the configuration-defined default value for the environment variable *PATH*. Since *PATH* can be modified by the user to include directories that could contain utilities replacing the standard utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x, applications need a way to determine the system-supplied *PATH* environment variable value that contains the correct search path for the standard utilities.

An application could use:

```
confstr(name, (char *)NULL, (size_t)0)
```

to find out how big a buffer is needed for the string value; use *malloc*( ) to allocate a buffer to hold the string; and call *confstr*( ) again to get the string. Alternately, it could allocate a fixed, static buffer that is big enough to hold most answers (perhaps 512 or 1 024 bytes), but then use *malloc*( ) to allocate a larger buffer if it finds that this is too small.

RATIONALE

Application developers can normally determine any configuration variable by means of reading from the stream opened by a call to:

```
popen("command -p getconf variable", "r");
```

The *confstr*( ) function with a *name* argument of _CS_PATH returns a string that can be used as a *PATH* environment variable setting that will reference the standard shell and utilities as described in the Shell and Utilities volume of IEEE Std 1003.1-200x.

The *confstr*( ) function copies the returned string into a buffer supplied by the application instead of returning a pointer to a string. This allows a cleaner function in some implementations (such as those with lightweight threads) and resolves questions about when the application must copy the string returned.

FUTURE DIRECTIONS

None.

SEE ALSO

pathconf( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *c99*

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

A table indicating the permissible values of *name* are added to the DESCRIPTION. All those marked EX are new in this issue.

Issue 6

The Open Group Corrigendum U033/7 is applied. The return value for the case returning the size of the buffer now explicitly states that this includes the terminating null.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated with new arguments which can be used to determine configuration strings for C compiler flags, linker/loader flags, and libraries for each different supported programming environment. This is a change to support data size neutrality.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The DESCRIPTION is updated to include text describing how _CS_PATH can be used to obtain a *PATH* to access the standard utilities.

The macros associated with the *c89* programming models are marked LEGACY and new equivalent macros associated with *c99* are introduced.

NAME

conj, conjf, conjl — complex conjugate functions

SYNOPSIS

```
#include <complex.h>

double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex conjugate of *z*, by reversing the sign of its imaginary part.

RETURN VALUE

These functions return the complex conjugate value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

carg( ), *cimag*( ), *cproj*( ), *creal*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

7339 **NAME**

7340 connect — connect a socket

7341 **SYNOPSIS**

7342 `#include <sys/socket.h>`

7343 `int connect(int socket, const struct sockaddr *address,`
7344 `socklen_t address_len);`

7345 **DESCRIPTION**

7346 The *connect*( ) function shall attempt to make a connection on a socket. The function takes the |
7347 following arguments:

7348 *socket* Specifies the file descriptor associated with the socket.

7349 *address* Points to a **sockaddr** structure containing the peer address. The length and
7350 format of the address depend on the address family of the socket.

7351 *address_len* Specifies the length of the **sockaddr** structure pointed to by the *address*
7352 argument.

7353 If the socket has not already been bound to a local address, *connect*( ) shall bind it to an address
7354 which, unless the socket's address family is AF_UNIX, is an unused local address.

7355 If the initiating socket is not connection-mode, then *connect*( ) shall set the socket's peer address,
7356 and no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all
7357 datagrams are sent on subsequent *send*( ) functions, and limits the remote sender for subsequent
7358 *recv*( ) functions. If *address* is a null address for the protocol, the socket's peer address shall be
7359 reset.

7360 If the initiating socket is connection-mode, then *connect*( ) shall attempt to establish a connection |
7361 to the address specified by the *address* argument. If the connection cannot be established |
7362 immediately and O_NONBLOCK is not set for the file descriptor for the socket, *connect*( ) shall |
7363 block for up to an unspecified timeout interval until the connection is established. If the timeout
7364 interval expires before the connection is established, *connect*( ) shall fail and the connection
7365 attempt shall be aborted. If *connect*( ) is interrupted by a signal that is caught while blocked
7366 waiting to establish a connection, *connect*( ) shall fail and set *errno* to [EINTR], but the connection
7367 request shall not be aborted, and the connection shall be established asynchronously.

7368 If the connection cannot be established immediately and O_NONBLOCK is set for the file
7369 descriptor for the socket, *connect*( ) shall fail and set *errno* to [EINPROGRESS], but the connection
7370 request shall not be aborted, and the connection shall be established asynchronously.
7371 Subsequent calls to *connect*( ) for the same socket, before the connection is established, shall fail
7372 and set *errno* to [EALREADY].

7373 When the connection has been established asynchronously, *select*( ) and *poll*( ) shall indicate that
7374 the file descriptor for the socket is ready for writing.

7375 The socket in use may require the process to have appropriate privileges to use the *connect*( )
7376 function.

7377 **RETURN VALUE**

7378 Upon successful completion, *connect*( ) shall return 0; otherwise, –1 shall be returned and *errno*
7379 set to indicate the error.

7380 **ERRORS**

7381 The *connect*( ) function shall fail if:

7382 [EADDRNOTAVAIL]
7383 The specified address is not available from the local machine.

[EAFNOSUPPORT]
The specified address is not a valid address for the address family of the specified socket.

[EALREADY] A connection request is already in progress for the specified socket.

[EBADF] The *socket* argument is not a valid file descriptor.

[ECONNREFUSED]
The target address was not listening for connections or refused the connection request.

[EINPROGRESS] O_NONBLOCK is set for the file descriptor for the socket and the connection cannot be immediately established; the connection shall be established asynchronously.

[EINTR] The attempt to establish a connection was interrupted by delivery of a signal that was caught; the connection shall be established asynchronously.

[EISCONN] The specified socket is connection-mode and is already connected.

[ENETUNREACH]
No route to the network is present.

[ENOTSOCK] The *socket* argument does not refer to a socket.

[EPROTOTYPE] The specified address has a different type than the socket bound to the specified peer address.

[ETIMEDOUT] The attempt to connect timed out before a connection was made.

If the address family of the socket is AF_UNIX, then *connect*( ) shall fail if:

[EIO] An I/O error occurred while reading from or writing to the file system.

[ELOOP] A loop exists in symbolic links encountered during resolution of the pathname in *address*.

[ENAMETOOLONG]
A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.

[ENOENT] A component of the pathname does not name an existing file or the pathname is an empty string.

[ENOTDIR] A component of the path prefix of the pathname in *address* is not a directory.

The *connect*( ) function may fail if:

[EACCES] Search permission is denied for a component of the path prefix; or write access to the named socket is denied.

[EADDRINUSE] Attempt to establish a connection that uses addresses that are already in use.

[ECONNRESET] Remote host reset the connection request.

[EHOSTUNREACH]
The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).

[EINVAL] The *address_len* argument is not a valid length for the address family; or invalid address family in the **sockaddr** structure.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in *address*.

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

[ENETDOWN] The local network interface used to reach the destination is down.

[ENOBUFS] No buffer space is available.

[EOPNOTSUPP] The socket is listening and cannot be connected.

EXAMPLES

None.

APPLICATION USAGE

If *connect*( ) fails, the state of the socket is unspecified. Conforming applications should close the file descriptor and create a new socket before attempting to reconnect.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

accept( ), *bind*( ), *close*( ), *getsockname*( ), *poll*( ), *select*( ), *send*( ), *shutdown*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

copysign, copysignf, copysignl — number manipulation function

SYNOPSIS

```
#include <math.h>

double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall produce a value with the magnitude of *x* and the sign of *y*. On implementations that represent a signed zero but do not treat negative zero consistently in arithmetic operations, these functions regard the sign of zero as positive.

RETURN VALUE

Upon successful completion, these functions shall return a value with the magnitude of *x* and the sign of *y*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

signbit( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

cos, cosf, cosl — cosine function

SYNOPSIS

```
#include <math.h>

double cos(double x);
float cosf(float x);
long double cosl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the cosine of their argument *x*, measured in radians.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the cosine of *x*.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, the value 1.0 shall be returned.

If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS

These functions shall fail if:

MX

Domain Error	The *x* argument is ±Inf.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

EXAMPLES

Taking the Cosine of a 45-Degree Angle

```
#include <math.h>
...
double radians = 45 * M_PI / 180;
double result;
...
result = cos(radians);
```

APPLICATION USAGE

These functions may lose accuracy when their argument is near an odd multiple of π/2 or is far from 0.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

acos( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *sin*( ), *tan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *cosf*( ) and *cosl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

NAME

cosf — cosine function

SYNOPSIS

```
#include <math.h>

float cosf(float x);
```

DESCRIPTION

Refer to *cos*( ).

NAME

cosh, coshf, coshl — hyperbolic cosine functions

SYNOPSIS

```
#include <math.h>

double cosh(double x);
float coshf(float x);
long double coshl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the hyperbolic cosine of their argument *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the hyperbolic cosine of *x*.

If the correct value would cause overflow, a range error shall occur and *cosh*( ), *coshf*( ), and *coshl*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, the value 1.0 shall be returned.

If *x* is ±Inf, +Inf shall be returned.

ERRORS

These functions shall fail if:

Range Error — The result would cause an overflow.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

For IEEE Std 754-1985 **double**, 710.5 < |*x*| implies that *cosh*(*x*) has overflowed.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

acosh( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *sinh*( ), *tanh*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *coshf*( ) and *coshl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

7603 **NAME**

7604 cosl — cosine function

7605 **SYNOPSIS**

7606 `#include <math.h>`

7607 `long double cosl(long double x);`

7608 **DESCRIPTION**

7609 Refer to *cos*( ).

NAME

cpow, cpowf, cpowl — complex power functions

SYNOPSIS

```
#include <complex.h>

double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x,
    long double complex y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex power function x^y, with a branch cut for the first parameter along the negative real axis.

RETURN VALUE

These functions shall return the complex power function value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cabs( ), *csqrt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

7640 **NAME**

7641 cproj, cprojf, cprojl — complex projection functions

7642 **SYNOPSIS**

7643
```
#include <complex.h>
```

7644
```
double complex cproj(double complex z);
```
7645
```
float complex cprojf(float complex z);
```
7646
```
long double complex cprojl(long double complex z);
```

7647 **DESCRIPTION**

7648 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7649 conflict between the requirements described here and the ISO C standard is unintentional. This
7650 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

7651 These functions shall compute a projection of *z* onto the Riemann sphere: *z* projects to *z*, except
7652 that all complex infinities (even those with one infinite part and one NaN part) project to
7653 positive infinity on the real axis. If *z* has an infinite part, then *cproj*(*z*) shall be equivalent to:

7654
```
INFINITY + I * copysign(0.0, cimag(z))
```

7655 **RETURN VALUE**

7656 These functions shall return the value of the projection onto the Riemann sphere.

7657 **ERRORS**

7658 No errors are defined.

7659 **EXAMPLES**

7660 None.

7661 **APPLICATION USAGE**

7662 None.

7663 **RATIONALE**

7664 Two topologies are commonly used in complex mathematics: the complex plane with its
7665 continuum of infinities, and the Riemann sphere with its single infinity. The complex plane is
7666 better suited for transcendental functions, the Riemann sphere for algebraic functions. The
7667 complex types with their multiplicity of infinities provide a useful (though imperfect) model for
7668 the complex plane. The *cproj*() function helps model the Riemann sphere by mapping all
7669 infinities to one, and should be used just before any operation, especially comparisons, that
7670 might give spurious results for any of the other infinities. Note that a complex value with one
7671 infinite part and one NaN part is regarded as an infinity, not a NaN, because if one part is
7672 infinite, the complex value is infinite independent of the value of the other part. For the same
7673 reason, *cabs*() returns an infinity if its argument has an infinite part and a NaN part.

7674 **FUTURE DIRECTIONS**

7675 None.

7676 **SEE ALSO**

7677 *carg*(), *cimag*(), *conj*(), *creal*(), the Base Definitions volume of IEEE Std 1003.1-200x,
7678 **<complex.h>**

7679 **CHANGE HISTORY**

7680 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7681 **NAME**

7682 creal, crealf, creall — complex real functions

7683 **SYNOPSIS**

7684
```
#include <complex.h>
```

7685
```
double creal(double complex z);
```
7686
```
float crealf(float complex z);
```
7687
```
long double creall(long double complex z);
```

7688 **DESCRIPTION**

7689 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7690 conflict between the requirements described here and the ISO C standard is unintentional. This
7691 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

7692 These functions shall compute the real part of *z*.

7693 **RETURN VALUE**

7694 These functions shall return the real part value.

7695 **ERRORS**

7696 No errors are defined.

7697 **EXAMPLES**

7698 None.

7699 **APPLICATION USAGE**

7700 For a variable *z* of complex type:

7701
```
z == creal(z) + cimag(z)*I
```

7702 **RATIONALE**

7703 None.

7704 **FUTURE DIRECTIONS**

7705 None.

7706 **SEE ALSO**

7707 *carg*( ), *cimag*( ), *conj*( ), *cproj*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
7708 **<complex.h>**

7709 **CHANGE HISTORY**

7710 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

7711 **NAME**

7712 creat — create a new file or rewrite an existing one

7713 **SYNOPSIS**

7714 OH
```
#include <sys/stat.h>
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```

7717 **DESCRIPTION**

7718 The function call:

```
creat(path, mode)
```

7720 shall be equivalent to:

```
open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

7722 **RETURN VALUE**

7723 Refer to *open*( ).

7724 **ERRORS**

7725 Refer to *open*( ).

7726 **EXAMPLES**

7727 **Creating a File**

7728 The following example creates the file **/tmp/file** with read and write permissions for the file
7729 owner and read permission for group and others. The resulting file descriptor is assigned to the
7730 *fd* variable.

```
#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *filename = "/tmp/file";
...
fd = creat(filename, mode);
...
```

7739 **APPLICATION USAGE**

7740 None.

7741 **RATIONALE**

7742 The *creat*( ) function is redundant. Its services are also provided by the *open*( ) function. It has
7743 been included primarily for historical purposes since many existing applications depend on it. It
7744 is best considered a part of the C binding rather than a function that should be provided in other
7745 languages.

7746 **FUTURE DIRECTIONS**

7747 None.

7748 **SEE ALSO**

7749 *open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<sys/stat.h>**,
7750 **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

NAME

crypt — string encoding function (**CRYPT**)

SYNOPSIS

XSI

```
#include <unistd.h>

char *crypt(const char *key, const char *salt);
```

DESCRIPTION

The *crypt*( ) function is a string encoding function. The algorithm is implementation-defined.

The *key* argument points to a string to be encoded. The *salt* argument is a string chosen from the set:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 . /
```

The first two characters of this string may be used to perturb the encoding algorithm.

The return value of *crypt*( ) points to static data that is overwritten by each call.

The *crypt*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *crypt*( ) shall return a pointer to the encoded string. The first two characters of the returned value shall be those of the *salt* argument. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *crypt*( ) function shall fail if:

[ENOSYS] The functionality is not supported on this implementation.

EXAMPLES

Encoding Passwords

The following example finds a user database entry matching a particular user name and changes the current password to a new password. The *crypt*( ) function generates an encoded version of each password. The first call to *crypt*( ) produces an encoded version of the old password; that encoded password is then compared to the password stored in the user database. The second call to *crypt*( ) encodes the new password before it is stored.

The *putpwent*( ) function, used in the following example, is not part of IEEE Std 1003.1-200x.

```
#include <unistd.h>
#include <pwd.h>
#include <string.h>
#include <stdio.h>
...
int valid_change;
int pfd;  /* Integer for file descriptor returned by open(). */
FILE *fpfd;  /* File pointer for use in putpwent(). */
struct passwd *p;
char user[100];
char oldpasswd[100];
char newpasswd[100];
```

```
        char savepasswd[100];
        ...
        valid_change = 0;
        while ((p = getpwent()) != NULL) {
            /* Change entry if found. */
            if (strcmp(p->pw_name, user) == 0) {
                if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
                    strcpy(savepasswd, crypt(newpasswd, user));
                    p->pw_passwd = savepasswd;
                    valid_change = 1;
                }
                else {
                    fprintf(stderr, "Old password is not valid\n");
                }
            }
            /* Put passwd entry into ptmp. */
            putpwent(p, fpfd);
        }
```

APPLICATION USAGE

The values returned by this function need not be portable among XSI-conformant systems.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

encrypt( ), *setkey*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

7835 **NAME**

7836 csin, csinf, csinl — complex sine functions

7837 **SYNOPSIS**

7838
```
#include <complex.h>
```

7839
```
double complex csin(double complex z);
```
7840
```
float complex csinf(float complex z);
```
7841
```
long double complex csinl(long double complex z);
```

7842 **DESCRIPTION**

7843 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7844 conflict between the requirements described here and the ISO C standard is unintentional. This
7845 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

7846 These functions shall compute the complex sine of *z*.

7847 **RETURN VALUE**

7848 These functions shall return the complex sine value.

7849 **ERRORS**

7850 No errors are defined.

7851 **EXAMPLES**

7852 None.

7853 **APPLICATION USAGE**

7854 None.

7855 **RATIONALE**

7856 None.

7857 **FUTURE DIRECTIONS**

7858 None.

7859 **SEE ALSO**

7860 *casin*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

7861 **CHANGE HISTORY**

7862 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

7863 **NAME**

7864 csinf — complex sine functions

7865 **SYNOPSIS**

7866
```
#include <complex.h>
```

7867
```
float complex csinf(float complex z);
```

7868 **DESCRIPTION**

7869 Refer to *csin*( ).

7870 **NAME**

7871 csinh, csinhf, csinhl — complex hyperbolic sine functions

7872 **SYNOPSIS**

7873 ```
#include <complex.h>
```

7874 ```
double complex csinh(double complex z);
```
7875 ```
float complex csinhf(float complex z);
```
7876 ```
long double complex csinhl(long double complex z);
```

7877 **DESCRIPTION**

7878 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7879 conflict between the requirements described here and the ISO C standard is unintentional. This
7880 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

7881 These functions shall compute the complex hyperbolic sine of *z*.

7882 **RETURN VALUE**

7883 These functions shall return the complex hyperbolic sine value.

7884 **ERRORS**

7885 No errors are defined.

7886 **EXAMPLES**

7887 None.

7888 **APPLICATION USAGE**

7889 None.

7890 **RATIONALE**

7891 None.

7892 **FUTURE DIRECTIONS**

7893 None.

7894 **SEE ALSO**

7895 *casinh*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

7896 **CHANGE HISTORY**

7897 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

7898 **NAME**

7899 csinl — complex sine functions

7900 **SYNOPSIS**

7901
```
#include <complex.h>
```

7902
```
long double complex csinl(long double complex z);
```

7903 **DESCRIPTION**

7904 Refer to *csin*( ).

7905 **NAME**

7906 csqrt, csqrtf, csqrtl — complex square root functions

7907 **SYNOPSIS**

7908
```
#include <complex.h>
```

7909
```
double complex csqrt(double complex z);
```
7910
```
float complex csqrtf(float complex z);
```
7911
```
long double complex csqrtl(long double complex z);
```

7912 **DESCRIPTION**

7913 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7914 conflict between the requirements described here and the ISO C standard is unintentional. This
7915 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

7916 These functions shall compute the complex square root of *z*, with a branch cut along the
7917 negative real axis.

7918 **RETURN VALUE**

7919 These functions shall return the complex square root value, in the range of the right half-plane
7920 (including the imaginary axis).

7921 **ERRORS**

7922 No errors are defined.

7923 **EXAMPLES**

7924 None.

7925 **APPLICATION USAGE**

7926 None.

7927 **RATIONALE**

7928 None.

7929 **FUTURE DIRECTIONS**

7930 None.

7931 **SEE ALSO**

7932 *cabs*( ), *cpow*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

7933 **CHANGE HISTORY**

7934 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

7935 **NAME**

7936 ctan, ctanf, ctanl — complex tangent functions

7937 **SYNOPSIS**

7938
```
#include <complex.h>
```

7939
```
double complex ctan(double complex z);
```
7940
```
float complex ctanf(float complex z);
```
7941
```
long double complex ctanl(long double complex z);
```

7942 **DESCRIPTION**

7943 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7944 conflict between the requirements described here and the ISO C standard is unintentional. This
7945 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

7946 These functions shall compute the complex tangent of *z*.

7947 **RETURN VALUE**

7948 These functions shall return the complex tangent value.

7949 **ERRORS**

7950 No errors are defined.

7951 **EXAMPLES**

7952 None.

7953 **APPLICATION USAGE**

7954 None.

7955 **RATIONALE**

7956 None.

7957 **FUTURE DIRECTIONS**

7958 None.

7959 **SEE ALSO**

7960 *catan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

7961 **CHANGE HISTORY**

7962 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

7963 **NAME**

7964 ctanf — complex tangent functions

7965 **SYNOPSIS**

7966 `#include <complex.h>`

7967 `float complex ctanf(float complex z);`

7968 **DESCRIPTION**

7969 Refer to *ctan*( ).

7970 **NAME**

7971 ctanh, ctanhf, ctanhl — complex hyperbolic tangent functions

7972 **SYNOPSIS**

7973
```
#include <complex.h>
```

7974
```
double complex ctanh(double complex z);
```
7975
```
float complex ctanhf(float complex z);
```
7976
```
long double complex ctanhl(long double complex z);
```

7977 **DESCRIPTION**

7978 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7979 conflict between the requirements described here and the ISO C standard is unintentional. This
7980 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

7981 These functions shall compute the complex hyperbolic tangent of *z*.

7982 **RETURN VALUE**

7983 These functions shall return the complex hyperbolic tangent value.

7984 **ERRORS**

7985 No errors are defined.

7986 **EXAMPLES**

7987 None.

7988 **APPLICATION USAGE**

7989 None.

7990 **RATIONALE**

7991 None.

7992 **FUTURE DIRECTIONS**

7993 None.

7994 **SEE ALSO**

7995 *catanh*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

7996 **CHANGE HISTORY**

7997 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

7998 **NAME**

7999 ctanl — complex tangent functions

8000 **SYNOPSIS**

8001 `#include <complex.h>`

8002 `long double complex ctanl(long double complex z);`

8003 **DESCRIPTION**

8004 Refer to *ctan*( ).

8005 **NAME**

8006 ctermid — generate a pathname for controlling terminal

8007 **SYNOPSIS**

8008 CX
```
#include <stdio.h>

char *ctermid(char *s);
```
8009

8010

8011 **DESCRIPTION**

8012 The *ctermid*( ) function shall generate a string that, when used as a pathname, refers to the
8013 current controlling terminal for the current process. If *ctermid*( ) returns a pathname, access to the
8014 file is not guaranteed.

8015 If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS
8016 functions, it shall ensure that the *ctermid*( ) function is called with a non-NULL parameter.

8017 **RETURN VALUE**

8018 If *s* is a null pointer, the string shall be generated in an area that may be static (and therefore may
8019 be overwritten by each call), the address of which shall be returned. Otherwise, *s* is assumed to
8020 point to a character array of at least L_ctermid bytes; the string is placed in this array and the
8021 value of *s* shall be returned. The symbolic constant L_ctermid is defined in **<stdio.h>**, and shall
8022 have a value greater than 0.

8023 The *ctermid*( ) function shall return an empty string if the pathname that would refer to the
8024 controlling terminal cannot be determined, or if the function is unsuccessful.

8025 **ERRORS**

8026 No errors are defined.

8027 **EXAMPLES**

8028 **Determining the Controlling Terminal for the Current Process**

8029 The following example returns a pointer to a string that identifies the controlling terminal for the
8030 current process. The pathname for the terminal is stored in the array pointed to by the *ptr*
8031 argument, which has a size of L_ctermid bytes, as indicated by the *term* argument.

```
8032 #include <stdio.h>
8033 ...
8034 char term[L_ctermid];
8035 char *ptr;

8036 ptr = ctermid(term);
```

8037 **APPLICATION USAGE**

8038 The difference between *ctermid*( ) and *ttyname*( ) is that *ttyname*( ) must be handed a file
8039 descriptor and return a path of the terminal associated with that file descriptor, while *ctermid*( )
8040 returns a string (such as `"/dev/tty"`) that refers to the current controlling terminal if used as a
8041 pathname.

8042 **RATIONALE**

8043 L_ctermid must be defined appropriately for a given implementation and must be greater than
8044 zero so that array declarations using it are accepted by the compiler. The value includes the
8045 terminating null byte.

8046 Conforming applications that use threads cannot call *ctermid*( ) with NULL as the parameter if
8047 either _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS is defined. If *s* is not
8048 NULL, the *ctermid*( ) function generates a string that, when used as a pathname, refers to the

current controlling terminal for the current process. If *s* is NULL, the return value of *ctermid*( ) is undefined.

There is no additional burden on the programmer—changing to use a hypothetical thread-safe version of *ctermid*( ) along with allocating a buffer is more of a burden than merely allocating a buffer. Application code should not assume that the returned string is short, as some implementations have more than two pathname components before reaching a logical device name.

FUTURE DIRECTIONS

None.

SEE ALSO

ttyname( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

8066 **NAME**

8067 ctime, ctime_r — convert a time value to date and time string

8068 **SYNOPSIS**

8069 ```
#include <time.h>
```

8070 ```
char *ctime(const time_t *clock);
```

8071 TSF ```
char *ctime_r(const time_t *clock, char *buf);
```

8072

8073 **DESCRIPTION**

8074 CX For *ctime*(): The functionality described on this reference page is aligned with the ISO C |
8075 standard. Any conflict between the requirements described here and the ISO C standard is
8076 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

8077 The *ctime*() function shall convert the time pointed to by *clock*, representing time in seconds
8078 since the Epoch, to local time in the form of a string. It shall be equivalent to: |

8079 ```
asctime(localtime(clock))
```

8080 CX The *asctime*(), *ctime*(), *gmtime*(), and *localtime*() functions shall return values in one of two static |
8081 objects: a broken-down time structure and an array of **char**. Execution of any of the functions |
8082 may overwrite the information returned in either of these objects by any of the other functions.

8083 The *ctime*() function need not be reentrant. A function that is not required to be reentrant is not
8084 required to be thread-safe.

8085 TSF The *ctime_r*() function shall convert the calendar time pointed to by *clock* to local time in exactly
8086 the same form as *ctime*() and puts the string into the array pointed to by *buf* (which shall be at |
8087 least 26 bytes in size) and return *buf*. |

8088 Unlike *ctime*(), the thread-safe version *ctime_r*() is not required to set *tzname*.

8089 **RETURN VALUE**

8090 The *ctime*() function shall return the pointer returned by *asctime*() with that broken-down time
8091 as an argument.

8092 TSF Upon successful completion, *ctime_r*() shall return a pointer to the string pointed to by *buf*.
8093 When an error is encountered, a null pointer shall be returned.

8094 **ERRORS**

8095 No errors are defined.

8096 **EXAMPLES**

8097 None.

8098 **APPLICATION USAGE**

8099 Values for the broken-down time structure can be obtained by calling *gmtime*() or *localtime*().
8100 The *ctime*() function is included for compatibility with older implementations, and does not
8101 support localized date and time formats. Applications should use the *strftime*() function to
8102 achieve maximum portability.

8103 The *ctime_r*() function is thread-safe and shall return values in a user-supplied buffer instead of
8104 possibly using a static data area that may be overwritten by each call.

8105 **RATIONALE**

8106 None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime( ), *clock*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

The *ctime_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *ctime*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

Extensions beyond the ISO C standard are now marked.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

8124 **NAME**

8125 daylight — daylight savings time flag

8126 **SYNOPSIS**

8127 XSI
```
#include <time.h>
```

8128
```
extern int daylight;
```

8129

8130 **DESCRIPTION**

8131 Refer to *tzset*( ).

NAME

dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch, dbm_firstkey, dbm_nextkey, dbm_open, dbm_store — database functions

SYNOPSIS

XSI
```
#include <ndbm.h>

int dbm_clearerr(DBM *db);
void dbm_close(DBM *db);
int dbm_delete(DBM *db, datum key);
int dbm_error(DBM *db);
datum dbm_fetch(DBM *db, datum key);
datum dbm_firstkey(DBM *db);
datum dbm_nextkey(DBM *db);
DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);
int dbm_store(DBM *db, datum key, datum content, int store_mode);
```

DESCRIPTION

These functions create, access, and modify a database.

A **datum** consists of at least two members, *dptr* and *dsize*. The *dptr* member points to an object that is *dsize* bytes in length. Arbitrary binary data, as well as character strings, may be stored in the object pointed to by *dptr*.

The database is stored in two files. One file is a directory containing a bit map of keys and has **.dir** as its suffix. The second file contains all data and has **.pag** as its suffix.

The *dbm_open*( ) function shall open a database. The *file* argument to the function is the pathname of the database. The function opens two files named *file***.dir** and *file***.pag**. The *open_flags* argument has the same meaning as the *flags* argument of *open*( ) except that a database opened for write-only access opens the files for read and write access and the behavior of the O_APPEND flag is unspecified. The *file_mode* argument has the same meaning as the third argument of *open*( ).

The *dbm_close*( ) function shall close a database. The application shall ensure that argument *db* is a pointer to a **dbm** structure that has been returned from a call to *dbm_open*( ).

These database functions shall support an internal block size large enough to support key/content pairs of at least 1 023 bytes.

The *dbm_fetch*( ) function shall read a record from a database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ). The argument *key* is a **datum** that has been initialized by the application to the value of the key that matches the key of the record the program is fetching.

The *dbm_store*( ) function shall write a record to a database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ). The argument *key* is a **datum** that has been initialized by the application to the value of the key that identifies (for subsequent reading, writing, or deleting) the record the application is writing. The argument *content* is a **datum** that has been initialized by the application to the value of the record the program is writing. The argument *store_mode* controls whether *dbm_store*( ) replaces any pre-existing record that has the same key that is specified by the *key* argument. The application shall set *store_mode* to either DBM_INSERT or DBM_REPLACE. If the database contains a record that matches the *key* argument and *store_mode* is DBM_REPLACE, the existing record shall be replaced with the new record. If the database contains a record that matches the *key* argument and *store_mode* is DBM_INSERT, the existing record shall be left unchanged and the new record

ignored. If the database does not contain a record that matches the *key* argument and *store_mode* is either DBM_INSERT or DBM_REPLACE, the new record shall be inserted in the database.

If the sum of a key/content pair exceeds the internal block size, the result is unspecified. Moreover, the application shall ensure that all key/content pairs that hash together fit on a single block. The *dbm_store*( ) function shall return an error in the event that a disk block fills with inseparable data.

The *dbm_delete*( ) function shall delete a record and its key from the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ). The argument *key* is a **datum** that has been initialized by the application to the value of the key that identifies the record the program is deleting.

The *dbm_firstkey*( ) function shall return the first key in the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ).

The *dbm_nextkey*( ) function shall return the next key in the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ). The application shall ensure that the *dbm_firstkey*( ) function is called before calling *dbm_nextkey*( ). Subsequent calls to *dbm_nextkey*( ) return the next key until all of the keys in the database have been returned.

The *dbm_error*( ) function shall return the error condition of the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ).

The *dbm_clearerr*( ) function shall clear the error condition of the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ).

The *dptr* pointers returned by these functions may point into static storage that may be changed by subsequent calls.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *dbm_store*( ) and *dbm_delete*( ) functions shall return 0 when they succeed and a negative value when they fail.

The *dbm_store*( ) function shall return 1 if it is called with a *flags* value of DBM_INSERT and the function finds an existing record with the same key.

The *dbm_error*( ) function shall return 0 if the error condition is not set and return a non-zero value if the error condition is set.

The return value of *dbm_clearerr*( ) is unspecified.

The *dbm_firstkey*( ) and *dbm_nextkey*( ) functions shall return a key **datum**. When the end of the database is reached, the *dptr* member of the key is a null pointer. If an error is detected, the *dptr* member of the key shall be a null pointer and the error condition of the database shall be set.

The *dbm_fetch*( ) function shall return a content **datum**. If no record in the database matches the key or if an error condition has been detected in the database, the *dptr* member of the content shall be a null pointer.

The *dbm_open*( ) function shall return a pointer to a database structure. If an error is detected during the operation, *dbm_open*( ) shall return a (**DBM ***)0.

8220 **ERRORS**

8221 No errors are defined.

8222 **EXAMPLES**

8223 None.

8224 **APPLICATION USAGE**

8225 The following code can be used to traverse the database:

8226 `for(key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))`

8227 The *dbm_* functions provided in this library should not be confused in any way with those of a
8228 general-purpose database management system. These functions do not provide for multiple
8229 search keys per entry, they do not protect against multi-user access (in other words they do not
8230 lock records or files), and they do not provide the many other useful database functions that are
8231 found in more robust database management systems. Creating and updating databases by use of
8232 these functions is relatively slow because of data copies that occur upon hash collisions. These
8233 functions are useful for applications requiring fast lookup of relatively static information that is
8234 to be indexed by a single key.

8235 The *dbm_delete*( ) function need not physically reclaim file space, although it does make it
8236 available for reuse by the database.

8237 After calling *dbm_store*( ) or *dbm_delete*( ) during a pass through the keys by *dbm_firstkey*( ) and
8238 *dbm_nextkey*( ), the application should reset the database by calling *dbm_firstkey*( ) before again
8239 calling *dbm_nextkey*( ). The contents of these files are unspecified and may not be portable.

8240 **RATIONALE**

8241 None.

8242 **FUTURE DIRECTIONS**

8243 None.

8244 **SEE ALSO**

8245 *open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ndbm.h>**

8246 **CHANGE HISTORY**

8247 First released in Issue 4, Version 2.

8248 **Issue 5**

8249 Moved from X/OPEN UNIX extension to BASE.

8250 Normative text previously in the APPLICATION USAGE section is moved to the
8251 DESCRIPTION.

8252 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

8253 **Issue 6**

8254 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

8255 **NAME**

8256 difftime — compute the difference between two calendar time values

8257 **SYNOPSIS**

8258 `#include <time.h>`

8259 `double difftime(time_t time1, time_t time0);`

8260 **DESCRIPTION**

8261 CX The functionality described on this reference page is aligned with the ISO C standard. Any
8262 conflict between the requirements described here and the ISO C standard is unintentional. This
8263 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

8264 The *difftime*( ) function shall compute the difference between two calendar times (as returned by
8265 *time*( )): *time1*– *time0*.

8266 **RETURN VALUE**

8267 The *difftime*( ) function shall return the difference expressed in seconds as a type **double**.

8268 **ERRORS**

8269 No errors are defined.

8270 **EXAMPLES**

8271 None.

8272 **APPLICATION USAGE**

8273 None.

8274 **RATIONALE**

8275 None.

8276 **FUTURE DIRECTIONS**

8277 None.

8278 **SEE ALSO**

8279 *asctime*( ), *clock*( ), *ctime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ),
8280 the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

8281 **CHANGE HISTORY**

8282 First released in Issue 4. Derived from the ISO C standard.

NAME

dirname — report the parent directory name of a file pathname

SYNOPSIS

XSI

```
#include <libgen.h>

char *dirname(char *path);
```

DESCRIPTION

The *dirname*( ) function shall take a pointer to a character string that contains a pathname, and return a pointer to a string that is a pathname of the parent directory of that file. Trailing '//' characters in the path are not counted as part of the path.

If *path* does not contain a '/', then *dirname*( ) shall return a pointer to the string ".". If *path* is a null pointer or points to an empty string, *dirname*( ) shall return a pointer to the string ".".

The *dirname*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *dirname*( ) function shall return a pointer to a string that is the parent directory of *path*. If *path* is a null pointer or points to an empty string, a pointer to a string "." is returned.

The *dirname*( ) function may modify the string pointed to by *path*, and may return a pointer to static storage that may then be overwritten by subsequent calls to *dirname*( ).

ERRORS

No errors are defined.

EXAMPLES

The following code fragment reads a pathname, changes the current working directory to the parent directory, and opens the file.

```
char path[MAXPATHLEN], *pathcopy;
int fd;
fgets(path, MAXPATHLEN, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

Sample Input and Output Strings for dirname( )

In the following table, the input string is the value pointed to by *path*, and the output string is the return value of the *dirname*( ) function.

Input String	Output String
"/usr/lib"	"/usr"
"/usr/"	"/"
"usr"	"."
"/"	"/"
"."	"."
".."	"."

Changing the Current Directory to the Parent Directory

The following program fragment reads a pathname, changes the current working directory to the parent directory, and opens the file.

```
#include <unistd.h>
#include <limits.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <libgen.h>
...
char path[PATH_MAX], *pathcopy;
int fd;
...
fgets(path, PATH_MAX, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

APPLICATION USAGE

The *dirname*( ) and *basename*( ) functions together yield a complete pathname. The expression *dirname*(*path*) obtains the pathname of the directory where *basename*(*path*) is found.

Since the meaning of the leading `"//"` is implementation-defined, *dirname*("**//foo**) may return either `"//"` or `'//'` (but nothing else).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

basename( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<libgen.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

8358 **NAME**

8359 div — compute the quotient and remainder of an integer division

8360 **SYNOPSIS**

8361
```
#include <stdlib.h>
```

8362
```
div_t div(int numer, int denom);
```

8363 **DESCRIPTION**

8364 CX The functionality described on this reference page is aligned with the ISO C standard. Any
8365 conflict between the requirements described here and the ISO C standard is unintentional. This
8366 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

8367 The *div*( ) function shall compute the quotient and remainder of the division of the numerator
8368 *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the integer
8369 of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be
8370 represented, the behavior is undefined; otherwise, *quot***denom*+*rem* shall equal *numer*.

8371 **RETURN VALUE**

8372 The *div*( ) function shall return a structure of type **div_t**, comprising both the quotient and the
8373 remainder. The structure includes the following members, in any order:

8374
```
int  quot;  /* quotient */
```
8375
```
int  rem;   /* remainder */
```

8376 **ERRORS**

8377 No errors are defined.

8378 **EXAMPLES**

8379 None.

8380 **APPLICATION USAGE**

8381 None.

8382 **RATIONALE**

8383 None.

8384 **FUTURE DIRECTIONS**

8385 None.

8386 **SEE ALSO**

8387 *ldiv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

8388 **CHANGE HISTORY**

8389 First released in Issue 4. Derived from the ISO C standard.

NAME

dlclose — close a *dlopen*( ) object

SYNOPSIS

XSI

```
#include <dlfcn.h>

int dlclose(void *handle);
```

DESCRIPTION

The *dlclose*( ) function shall inform the system that the object referenced by a *handle* returned from a previous *dlopen*( ) invocation is no longer needed by the application.

The use of *dlclose*( ) reflects a statement of intent on the part of the process, but does not create any requirement upon the implementation, such as removal of the code or symbols referenced by *handle*. Once an object has been closed using *dlclose*( ) an application should assume that its symbols are no longer available to *dlsym*( ). All objects loaded automatically as a result of invoking *dlopen*( ) on the referenced object shall also be closed if this is the last reference to it.

Although a *dlclose*( ) operation is not required to remove structures from an address space, neither is an implementation prohibited from doing so. The only restriction on such a removal is that no object shall be removed to which references have been relocated, until or unless all such references are removed. For instance, an object that had been loaded with a *dlopen*( ) operation specifying the RTLD_GLOBAL flag might provide a target for dynamic relocations performed in the processing of other objects—in such environments, an application may assume that no relocation, once made, shall be undone or remade unless the object requiring the relocation has itself been removed.

RETURN VALUE

If the referenced object was successfully closed, *dlclose*( ) shall return 0. If the object could not be closed, or if *handle* does not refer to an open object, *dlclose*( ) shall return a non-zero value. More detailed diagnostic information shall be available through *dlerror*( ).

ERRORS

No errors are defined.

EXAMPLES

The following example illustrates use of *dlopen*( ) and *dlclose*( ):

```
...
/* Open a dynamic library and then close it ... */

#include <dlfcn.h>
void *mylib;
int eret;

mylib = dlopen("mylib.so", RTLD_LOCAL | RTLD_LAZY);
...
eret = dlclose(mylib);
...
```

APPLICATION USAGE

A conforming application should employ a *handle* returned from a *dlopen*( ) invocation only within a given scope bracketed by the *dlopen*( ) and *dlclose*( ) operations. Implementations are free to use reference counting or other techniques such that multiple calls to *dlopen*( ) referencing the same object may return the same object for *handle*. Implementations are also free to reuse a *handle*. For these reasons, the value of a *handle* must be treated as an opaque object by the application, used only in calls to *dlsym*( ) and *dlclose*( ).

8436 **RATIONALE**

8437 None.

8438 **FUTURE DIRECTIONS**

8439 None.

8440 **SEE ALSO**

8441 *dlerror*( ), *dlopen*( ), *dlsym*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dlfcn.h>**

8442 **CHANGE HISTORY**

8443 First released in Issue 5.

8444 **Issue 6**

8445 The DESCRIPTION is updated to say that the referenced object is closed ‘‘if this is the last
8446 reference to it’’.

8447 **NAME**

8448 dlerror — get diagnostic information

8449 **SYNOPSIS**

8450 XSI
```
#include <dlfcn.h>
```
8451
```
char *dlerror(void);
```
8452

8453 **DESCRIPTION**

8454 The *dlerror*( ) function shall return a null-terminated character string (with no trailing <newline>)
8455 that describes the last error that occurred during dynamic linking processing. If no dynamic
8456 linking errors have occurred since the last invocation of *dlerror*( ), *dlerror*( ) shall return NULL.
8457 Thus, invoking *dlerror*( ) a second time, immediately following a prior invocation, shall result in
8458 NULL being returned.

8459 The *dlerror*( ) function need not be reentrant. A function that is not required to be reentrant is not
8460 required to be thread-safe.

8461 **RETURN VALUE**

8462 If successful, *dlerror*( ) shall return a null-terminated character string; otherwise, NULL shall be
8463 returned.

8464 **ERRORS**

8465 No errors are defined.

8466 **EXAMPLES**

8467 The following example prints out the last dynamic linking error:

```
8468 ...
8469 #include <dlfcn.h>
8470 char *errstr;
8471 errstr = dlerror();
8472 if (errstr != NULL)
8473 printf ("A dynamic linking error occurred: (%s)\n", errstr);
8474 ...
```

8475 **APPLICATION USAGE**

8476 The messages returned by *dlerror*( ) may reside in a static buffer that is overwritten on each call
8477 to *dlerror*( ). Application code should not write to this buffer. Programs wishing to preserve an
8478 error message should make their own copies of that message. Depending on the application
8479 environment with respect to asynchronous execution events, such as signals or other
8480 asynchronous computation sharing the address space, conforming applications should use a |
8481 critical section to retrieve the error pointer and buffer. |

8482 **RATIONALE**

8483 None.

8484 **FUTURE DIRECTIONS**

8485 None.

8486 **SEE ALSO**

8487 *dlclose*( ), *dlopen*( ), *dlsym*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dlfcn.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

In the DESCRIPTION the note about reentrancy and thread-safety is added.

8492 **NAME**

8493 dlopen — gain access to an executable object file

8494 **SYNOPSIS**

8495 XSI
```
#include <dlfcn.h>
```

8496
```
void *dlopen(const char *file, int mode);
```

8497

8498 **DESCRIPTION**

8499 The *dlopen*( ) function shall make an executable object file specified by *file* available to the calling
8500 program. The class of files eligible for this operation and the manner of their construction are
8501 implementation-defined, though typically such files are executable objects such as shared
8502 libraries, relocatable files, or programs. Note that some implementations permit the construction
8503 of dependencies between such objects that are embedded within files. In such cases, a *dlopen*( )
8504 operation shall load such dependencies in addition to the object referenced by *file*.
8505 Implementations may also impose specific constraints on the construction of programs that can
8506 employ *dlopen*( ) and its related services.

8507 A successful *dlopen*( ) shall return a *handle* which the caller may use on subsequent calls to
8508 *dlsym*( ) and *dlclose*( ). The value of this *handle* should not be interpreted in any way by the caller.

8509 The *file* argument is used to construct a pathname to the object file. If *file* contains a slash
8510 character, the *file* argument is used as the pathname for the file. Otherwise, *file* is used in an
8511 implementation-defined manner to yield a pathname.

8512 If the value of *file* is 0, *dlopen*( ) shall provide a *handle* on a global symbol object. This object shall
8513 provide access to the symbols from an ordered set of objects consisting of the original program
8514 image file, together with any objects loaded at program start-up as specified by that process
8515 image file (for example, shared libraries), and the set of objects loaded using a *dlopen*( ) operation
8516 together with the RTLD_GLOBAL flag. As the latter set of objects can change during execution,
8517 the set identified by *handle* can also change dynamically.

8518 Only a single copy of an object file is brought into the address space, even if *dlopen*( ) is invoked
8519 multiple times in reference to the file, and even if different pathnames are used to reference the
8520 file.

8521 The *mode* parameter describes how *dlopen*( ) shall operate upon *file* with respect to the processing
8522 of relocations and the scope of visibility of the symbols provided within *file*. When an object is
8523 brought into the address space of a process, it may contain references to symbols whose
8524 addresses are not known until the object is loaded. These references shall be relocated before the
8525 symbols can be accessed. The *mode* parameter governs when these relocations take place and
8526 may have the following values:

8527 RTLD_LAZY — Relocations shall be performed at an implementation-defined time, ranging from the time of the *dlopen*( ) call until the first reference to a given symbol occurs. Specifying RTLD_LAZY should improve performance on implementations supporting dynamic symbol binding as a process may not reference all of the functions in any given object. And, for systems supporting dynamic symbol resolution for normal process execution, this behavior mimics the normal handling of process execution.

8535 RTLD_NOW — All necessary relocations shall be performed when the object is first loaded. This may waste some processing if relocations are performed for functions that are never referenced. This behavior may be useful for applications that need to know as soon as an object is loaded that all

symbols referenced during execution are available.

Any object loaded by *dlopen*( ) that requires relocations against global symbols can reference the symbols in the original process image file, any objects loaded at program start-up, from the object itself as well as any other object included in the same *dlopen*( ) invocation, and any objects that were loaded in any *dlopen*( ) invocation and which specified the RTLD_GLOBAL flag. To determine the scope of visibility for the symbols loaded with a *dlopen*( ) invocation, the *mode* parameter should be a bitwise-inclusive OR with one of the following values:

RTLD_GLOBAL — The object's symbols shall be made available for the relocation processing of any other object. In addition, symbol lookup using *dlopen*(*0*, *mode*) and an associated *dlsym*( ) allows objects loaded with this *mode* to be searched.

RTLD_LOCAL — The object's symbols shall not be made available for the relocation processing of any other object.

If neither RTLD_GLOBAL nor RTLD_LOCAL are specified, then an implementation-defined default behavior shall be applied.

If a *file* is specified in multiple *dlopen*( ) invocations, *mode* is interpreted at each invocation. Note, however, that once RTLD_NOW has been specified all relocations shall have been completed rendering further RTLD_NOW operations redundant and any further RTLD_LAZY operations irrelevant. Similarly, note that once RTLD_GLOBAL has been specified the object shall maintain the RTLD_GLOBAL status regardless of any previous or future specification of RTLD_LOCAL, as long as the object remains in the address space (see *dlclose*( )).

Symbols introduced into a program through calls to *dlopen*( ) may be used in relocation activities. Symbols so introduced may duplicate symbols already defined by the program or previous *dlopen*( ) operations. To resolve the ambiguities such a situation might present, the resolution of a symbol reference to symbol definition is based on a symbol resolution order. Two such resolution orders are defined: *load* or *dependency* ordering. Load order establishes an ordering among symbol definitions, such that the definition first loaded (including definitions from the image file and any dependent objects loaded with it) has priority over objects added later (via *dlopen*( )). Load ordering is used in relocation processing. Dependency ordering uses a breadth-first order starting with a given object, then all of its dependencies, then any dependents of those, iterating until all dependencies are satisfied. With the exception of the global symbol object obtained via a *dlopen*( ) operation on a *file* of 0, dependency ordering is used by the *dlsym*( ) function. Load ordering is used in *dlsym*( ) operations upon the global symbol object.

When an object is first made accessible via *dlopen*( ) it and its dependent objects are added in dependency order. Once all the objects are added, relocations are performed using load order. Note that if an object or its dependencies had been previously loaded, the load and dependency orders may yield different resolutions.

The symbols introduced by *dlopen*( ) operations, and available through *dlsym*( ) are at a minimum those which are exported as symbols of global scope by the object. Typically such symbols shall be those that were specified in (for example) C source code as having *extern* linkage. The precise manner in which an implementation constructs the set of exported symbols for a *dlopen*( ) object is specified by that implementation.

RETURN VALUE

If *file* cannot be found, cannot be opened for reading, is not of an appropriate object format for processing by *dlopen*( ), or if an error occurs during the process of loading *file* or relocating its symbolic references, *dlopen*( ) shall return NULL. More detailed diagnostic information shall be available through *dlerror*( ).

8585 **ERRORS**

8586 No errors are defined.

8587 **EXAMPLES**

8588 None.

8589 **APPLICATION USAGE**

8590 None.

8591 **RATIONALE**

8592 None.

8593 **FUTURE DIRECTIONS**

8594 None.

8595 **SEE ALSO**

8596 *dlclose*( ), *dlerror*( ), *dlsym*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dlfcn.h>**

8597 **CHANGE HISTORY**

8598 First released in Issue 5.

8599 **NAME**

8600 dlsym — obtain the address of a symbol from a *dlopen*( ) object

8601 **SYNOPSIS**

8602 XSI
```
#include <dlfcn.h>
```

8603
```
void *dlsym(void *restrict handle, const char *restrict name);
```

8604

8605 **DESCRIPTION**

8606 The *dlsym*( ) function shall obtain the address of a symbol defined within an object made |
8607 accessible through a *dlopen*( ) call. The *handle* argument is the value returned from a call to |
8608 *dlopen*( ) (and which has not since been released via a call to *dlclose*( )), and *name* is the symbol's
8609 name as a character string.

8610 The *dlsym*( ) function shall search for the named symbol in all objects loaded automatically as a
8611 result of loading the object referenced by *handle* (see *dlopen*( )). Load ordering is used in *dlsym*( )
8612 operations upon the global symbol object. The symbol resolution algorithm used shall be
8613 dependency order as described in *dlopen*( ).

8614 The RTLD_NEXT flag is reserved for future use.

8615 **RETURN VALUE**

8616 If *handle* does not refer to a valid object opened by *dlopen*( ), or if the named symbol cannot be
8617 found within any of the objects associated with *handle*, *dlsym*( ) shall return NULL. More
8618 detailed diagnostic information shall be available through *dlerror*( ).

8619 **ERRORS**

8620 No errors are defined.

8621 **EXAMPLES**

8622 The following example shows how *dlopen*( ) and *dlsym*( ) can be used to access either function or
8623 data objects. For simplicity, error checking has been omitted.

```
8624     void    *handle;
8625     int     *iptr, (*fptr)(int);

8626     /* open the needed object */
8627     handle = dlopen("/usr/home/me/libfoo.so", RTLD_LOCAL | RTLD_LAZY);      |

8628     /* find the address of function and data objects */                     |
8629     fptr = (int (*)(int))dlsym(handle, "my_function");
8630     iptr = (int *)dlsym(handle, "my_object");

8631     /* invoke function, passing value of integer as a parameter */
8632     (*fptr)(*iptr);
```

8633 **APPLICATION USAGE**

8634 Special purpose values for *handle* are reserved for future use. These values and their meanings
8635 are:

8636 RTLD_DEFAULT The symbol lookup happens in the normal global scope; that is, a search for a |
8637 symbol using this handle would find the same definition as a direct use of this |
8638 symbol in the program code. |

8639 RTLD_NEXT Specifies the next object after this one that defines *name*. *This one* refers to the
8640 object containing the invocation of *dlsym*( ). The *next* object is the one found
8641 upon the application of a load order symbol resolution algorithm (see
8642 *dlopen*( )). The next object is either one of global scope (because it was
8643 introduced as part of the original process image or because it was added with

a *dlopen*( ) operation including the RTLD_GLOBAL flag), or is an object that was included in the same *dlopen*( ) operation that loaded this one.

The RTLD_NEXT flag is useful to navigate an intentionally created hierarchy of multiply-defined symbols created through *interposition*. For example, if a program wished to create an implementation of *malloc*( ) that embedded some statistics gathering about memory allocations, such an implementation could use the real *malloc*( ) definition to perform the memory allocation—and itself only embed the necessary logic to implement the statistics gathering function.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

dlclose( ), *dlerror*( ), *dlopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dlfcn.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

The **restrict** keyword is added to the *dlsym*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

8663 **NAME**

8664 drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, seed48, srand48 — generate
8665 uniformly distributed pseudo-random numbers

8666 **SYNOPSIS**

8667 XSI
```
#include <stdlib.h>

double drand48(void);
double erand48(unsigned short xsubi[3]);
long jrand48(unsigned short xsubi[3]);
void lcong48(unsigned short param[7]);
long lrand48(void);
long mrand48(void);
long nrand48(unsigned short xsubi[3]);
unsigned short *seed48(unsigned short seed16v[3]);
void srand48(long seedval);
```
8677

8678 **DESCRIPTION**

8679 This family of functions shall generate pseudo-random numbers using a linear congruential |
8680 algorithm and 48-bit integer arithmetic.

8681 The *drand48*( ) and *erand48*( ) functions shall return non-negative, double-precision, floating-
8682 point values, uniformly distributed over the interval [0.0,1.0).

8683 The *lrand48*( ) and *nrand48*( ) functions shall return non-negative, long integers, uniformly
8684 distributed over the interval $[0,2^{31})$.

8685 The *mrand48*( ) and *jrand48*( ) functions shall return signed long integers uniformly distributed
8686 over the interval $[-2^{31},2^{31})$.

8687 The *srand48*( ), *seed48*( ), and *lcong48*( ) are initialization entry points, one of which should be
8688 invoked before either *drand48*( ), *lrand48*( ), or *mrand48*( ) is called. (Although it is not
8689 recommended practice, constant default initializer values shall be supplied automatically if
8690 *drand48*( ), *lrand48*( ), or *mrand48*( ) is called without a prior call to an initialization entry point.)
8691 The *erand48*( ), *nrand48*( ), and *jrand48*( ) functions do not require an initialization entry point to
8692 be called first.

8693 All the routines work by generating a sequence of 48-bit integer values, X_i, according to the
8694 linear congruential formula:

8695 $$X_{n+1} = (aX_n + c)_{\mathrm{mod}\ m} \qquad n \geq 0$$

8696 The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless *lcong48*( ) is invoked,
8697 the multiplier value a and the addend value c are given by:

8698 $$a = 5\mathrm{DEECE66D}_{16} = 273673163155_8$$

8699 $$c = \mathrm{B}_{16} = 13_8$$

8700 The value returned by any of the *drand48*( ), *erand48*( ), *jrand48*( ), *lrand48*( ), *mrand48*( ), or
8701 *nrand48*( ) functions is computed by first generating the next 48-bit X_i in the sequence. Then the
8702 appropriate number of bits, according to the type of data item to be returned, are copied from
8703 the high-order (leftmost) bits of X_i and transformed into the returned value.

8704 The *drand48*( ), *lrand48*( ), and *mrand48*( ) functions store the last 48-bit X_i generated in an
8705 internal buffer; that is why the application shall ensure that these are initialized prior to being
8706 invoked. The *erand48*( ), *nrand48*( ), and *jrand48*( ) functions require the calling program to
8707 provide storage for the successive X_i values in the array specified as an argument when the

functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, *erand48*(), *nrand48*(), and *jrand48*() allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers; that is, the sequence of numbers in each stream shall *not* depend upon how many times the routines are called to generate numbers for the other streams.

The initializer function *srand48*() sets the high-order 32 bits of X_i to the low-order 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function *seed48*() sets the value of X_i to the 48-bit value specified in the argument array. The low-order 16 bits of X_i are set to the low-order 16 bits of *seed16v*[*0*]. The mid-order 16 bits of X_i are set to the low-order 16 bits of *seed16v*[*1*]. The high-order 16 bits of X_i are set to the low-order 16 bits of *seed16v*[*2*]. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by *seed48*(), and a pointer to this buffer is the value returned by *seed48*(). This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time—use the pointer to get at and store the last X_i value, and then use this value to reinitialize via *seed48*() when the program is restarted. |

The initializer function *lcong48*() allows the user to specify the initial X_i, the multiplier value *a*, and the addend value *c*. Argument array elements *param*[*0-2*] specify X_i, *param*[*3-5*] specify the multiplier *a*, and *param*[*6*] specifies the 16-bit addend *c*. After *lcong48*() is called, a subsequent call to either *srand48*() or *seed48*() shall restore the standard multiplier and addend values, *a* and *c*, specified above.

The *drand48*(), *lrand48*(), and *mrand48*() functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

As described in the DESCRIPTION above.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

rand(), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

8751 **NAME**

8752 dup, dup2 — duplicate an open file descriptor

8753 **SYNOPSIS**

8754 ```
#include <unistd.h>
```

8755 ```
int dup(int fildes);
```
8756 ```
int dup2(int fildes, int fildes2);
```

8757 **DESCRIPTION**

8758 The *dup*( ) and *dup2*( ) functions provide an alternative interface to the service provided by
8759 *fcntl*( ) using the F_DUPFD command. The call:

8760 ```
fid = dup(fildes);
```

8761 shall be equivalent to: |

8762 ```
fid = fcntl(fildes, F_DUPFD, 0);
```

8763 The call:

8764 ```
fid = dup2(fildes, fildes2);
```

8765 shall be equivalent to: |

8766 ```
close(fildes2);
```
8767 ```
fid = fcntl(fildes, F_DUPFD, fildes2);
```

8768 except for the following:

8769 • If *fildes2* is less than 0 or greater than or equal to {OPEN_MAX}, *dup2*( ) shall return –1 with
8770 *errno* set to [EBADF].

8771 • If *fildes* is a valid file descriptor and is equal to *fildes2*, *dup2*( ) shall return *fildes2* without
8772 closing it.

8773 • If *fildes* is not a valid file descriptor, *dup2*( ) shall return –1 and shall not close *fildes2*.

8774 • The value returned shall be equal to the value of *fildes2* upon successful completion, or –1
8775 upon failure.

8776 **RETURN VALUE**

8777 Upon successful completion a non-negative integer, namely the file descriptor, shall be returned;
8778 otherwise, –1 shall be returned and *errno* set to indicate the error.

8779 **ERRORS**

8780 The *dup*( ) function shall fail if:

8781 [EBADF] The *fildes* argument is not a valid open file descriptor.

8782 [EMFILE] The number of file descriptors in use by this process would exceed
8783 {OPEN_MAX}.

8784 The *dup2*( ) function shall fail if:

8785 [EBADF] The *fildes* argument is not a valid open file descriptor or the argument *fildes2* is
8786 negative or greater than or equal to {OPEN_MAX}.

8787 [EINTR] The *dup2*( ) function was interrupted by a signal.

EXAMPLES

Redirecting Standard Output to a File

The following example closes standard output for the current processes, re-assigns standard output to go to the file referenced by *pfd*, and closes the original file descriptor to clean up.

```
#include <unistd.h>
...
int pfd;
...
close(1);
dup(pfd);
close(pfd);
...
```

Redirecting Error Messages

The following example redirects messages from *stderr* to *stdout*.

```
#include <unistd.h>
...
dup2(1, 2);
...
```

APPLICATION USAGE

None.

RATIONALE

The *dup*( ) and *dup2*( ) functions are redundant. Their services are also provided by the *fcntl*( ) function. They have been included in this volume of IEEE Std 1003.1-200x primarily for historical reasons, since many existing applications use them.

While the brief code segment shown is very similar in behavior to *dup2*( ), a conforming implementation based on other functions defined in this volume of IEEE Std 1003.1-200x is significantly more complex. Least obvious is the possible effect of a signal-catching function that could be invoked between steps and allocate or deallocate file descriptors. This could be avoided by blocking signals.

The *dup2*( ) function is not marked obsolescent because it presents a type-safe version of functionality provided in a type-unsafe version by *fcntl*( ). It is used in the POSIX Ada binding.

The *dup2*( ) function is not intended for use in critical regions as a synchronization mechanism.

In the description of [EBADF], the case of *fildes* being out of range is covered by the given case of *fildes* not being valid. The descriptions for *fildes* and *fildes2* are different because the only kind of invalidity that is relevant for *fildes2* is whether it is out of range; that is, it does not matter whether *fildes2* refers to an open file when the *dup2*( ) call is made.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *fcntl*( ), *open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The RATIONALE section is added.

8832 **NAME**

8833 ecvt, fcvt, gcvt — convert a floating-point number to a string (**LEGACY**)

8834 **SYNOPSIS**

8835 XSI
```
#include <stdlib.h>

char *ecvt(double value, int ndigit, int *restrict decpt,
    int *restrict sign);
char *fcvt(double value, int ndigit, int *restrict decpt,
    int *restrict sign);
char *gcvt(double value, int ndigit, char *buf);
```
8841

8842 **DESCRIPTION**

8843 The *ecvt*( ), *fcvt*( ), and *gcvt*( ) functions shall convert floating-point numbers to null-terminated
8844 strings.

8845 The *ecvt*( ) function shall convert *value* to a null-terminated string of *ndigit* digits (where *ndigit* is
8846 reduced to an unspecified limit determined by the precision of a **double**) and return a pointer to |
8847 the string. The high-order digit shall be non-zero, unless the value is 0. The low-order digit shall |
8848 be rounded in an implementation-defined manner. The position of the radix character relative to |
8849 the beginning of the string shall be stored in the integer pointed to by *decpt* (negative means to |
8850 the left of the returned digits). If *value* is zero, it is unspecified whether the integer pointed to by
8851 *decpt* would be 0 or 1. The radix character shall not be included in the returned string. If the sign |
8852 of the result is negative, the integer pointed to by *sign* shall be non-zero; otherwise, it shall be 0. |

8853 If the converted value is out of range or is not representable, the contents of the returned string
8854 are unspecified.

8855 The *fcvt*( ) function shall be equivalent to *ecvt*( ), except that *ndigit* specifies the number of digits |
8856 desired after the radix character. The total number of digits in the result string is restricted to an
8857 unspecified limit as determined by the precision of a **double**.

8858 The *gcvt*( ) function shall convert *value* to a null-terminated string (similar to that of the %g
8859 conversion specification format of *printf*( )) in the array pointed to by *buf* and shall return *buf*. It |
8860 shall produce *ndigit* significant digits (limited to an unspecified value determined by the |
8861 precision of a **double**) in the %f conversion specification format of *printf*( ) if possible, or the %e
8862 conversion specification format of *printf*( ) (scientific notation) otherwise. A minus sign shall be |
8863 included in the returned string if *value* is less than 0. A radix character shall be included in the |
8864 returned string if *value* is not a whole number. Trailing zeros shall be suppressed where *value* is |
8865 not a whole number. The radix character is determined by the current locale. If *setlocale*( ) has not
8866 been called successfully, the default locale, POSIX, is used. The default locale specifies a period
8867 ('.') as the radix character. The *LC_NUMERIC* category determines the value of the radix
8868 character within the current locale.

8869 These functions need not be reentrant. A function that is not required to be reentrant is not
8870 required to be thread-safe.

8871 **RETURN VALUE**

8872 The *ecvt*( ) and *fcvt*( ) functions shall return a pointer to a null-terminated string of digits.

8873 The *gcvt*( ) function shall return *buf*.

8874 The return values from *ecvt*( ) and *fcvt*( ) may point to static data which may be overwritten by
8875 subsequent calls to these functions.

8876 **ERRORS**

8877 No errors are defined.

8878 **EXAMPLES**

8879 None.

8880 **APPLICATION USAGE**

8881 *sprintf*( ) is preferred over this function.

8882 **RATIONALE**

8883 None.

8884 **FUTURE DIRECTIONS**

8885 These functions may be withdrawn in a future version.

8886 **SEE ALSO**

8887 *printf*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

8888 **CHANGE HISTORY**

8889 First released in Issue 4, Version 2.

8890 **Issue 5**

8891 Moved from X/OPEN UNIX extension to BASE.

8892 Normative text previously in the APPLICATION USAGE section is moved to the
8893 DESCRIPTION.

8894 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

8895 **Issue 6**

8896 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

8897 This function is marked LEGACY.

8898 The **restrict** keyword is added to the *ecvt*( ) and *fcvt*( ) prototypes for alignment with the
8899 ISO/IEC 9899: 1999 standard.

8900 The DESCRIPTION is updated to explicitly use ‘‘conversion specification’’ to describe %g, %f,
8901 and %e.

8902 **NAME**

8903 encrypt — encoding function (**CRYPT**)

8904 **SYNOPSIS**

8905 XSI
```
#include <unistd.h>
```

8906
```
void encrypt(char block[64], int edflag);
```

8907

8908 **DESCRIPTION**

8909 The *encrypt*( ) function shall provide access to an implementation-defined encoding algorithm. |
8910 The key generated by *setkey*( ) is used to encrypt the string *block* with *encrypt*( ). |

8911 The *block* argument to *encrypt*( ) shall be an array of length 64 bytes containing only the bytes |
8912 with values of 0 and 1. The array is modified in place to a similar array using the key set by |
8913 *setkey*( ). If *edflag* is 0, the argument is encoded. If *edflag* is 1, the argument may be decoded (see
8914 the APPLICATION USAGE section); if the argument is not decoded, *errno* shall be set to
8915 [ENOSYS].

8916 The *encrypt*( ) function shall not change the setting of *errno* if successful. An application wishing
8917 to check for error situations should set *errno* to 0 before calling *encrypt*( ). If *errno* is non-zero on
8918 return, an error has occurred.

8919 The *encrypt*( ) function need not be reentrant. A function that is not required to be reentrant is
8920 not required to be thread-safe.

8921 **RETURN VALUE**

8922 The *encrypt*( ) function shall not return a value.

8923 **ERRORS**

8924 The *encrypt*( ) function shall fail if:

8925 [ENOSYS] The functionality is not supported on this implementation.

8926 **EXAMPLES**

8927 None.

8928 **APPLICATION USAGE**

8929 Historical implementations of the *encrypt*( ) function used a rather primitive encoding algorithm. |

8930 In some environments, decoding might not be implemented. This is related to some Government |
8931 restrictions on encryption and decryption routines. Historical practice has been to ship a |
8932 different version of the encryption library without the decryption feature in the routines |
8933 supplied. Thus the exported version of *encrypt*( ) does encoding but not decoding.

8934 **RATIONALE**

8935 None.

8936 **FUTURE DIRECTIONS**

8937 None.

8938 **SEE ALSO**

8939 *crypt*( ), *setkey*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

8940 **CHANGE HISTORY**

8941 First released in Issue 1. Derived from Issue 1 of the SVID.

8942 **Issue 5**

8943 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

8944 **Issue 6**

8945 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

8946 **NAME**

8947 endgrent, getgrent, setgrent — group database entry functions

8948 **SYNOPSIS**

8949 XSI
```
#include <grp.h>

void endgrent(void);
struct group *getgrent(void);
void setgrent(void);
```

8953

8954 **DESCRIPTION**

8955 The *getgrent*( ) function shall return a pointer to a structure containing the broken-out fields of an
8956 entry in the group database. When first called, *getgrent*( ) shall return a pointer to a **group**
8957 structure containing the first entry in the group database. Thereafter, it shall return a pointer to a
8958 **group** structure containing the next group structure in the group database, so successive calls
8959 may be used to search the entire database.

8960 An implementation that provides extended security controls may impose further
8961 implementation-defined restrictions on accessing the group database. In particular, the system
8962 may deny the existence of some or all of the group database entries associated with groups other
8963 than those groups associated with the caller and may omit users other than the caller from the
8964 list of members of groups in database entries that are returned.

8965 The *setgrent*( ) function shall rewind the group database to allow repeated searches.

8966 The *endgrent*( ) function may be called to close the group database when processing is complete.

8967 These functions need not be reentrant. A function that is not required to be reentrant is not
8968 required to be thread-safe.

8969 **RETURN VALUE**

8970 When first called, *getgrent*( ) shall return a pointer to the first group structure in the group
8971 database. Upon subsequent calls it shall return the next group structure in the group database.
8972 The *getgrent*( ) function shall return a null pointer on end-of-file or an error and *errno* may be set
8973 to indicate the error.

8974 The return value may point to a static area which is overwritten by a subsequent call to
8975 *getgrgid*( ), *getgrnam*( ), or *getgrent*( ).

8976 **ERRORS**

8977 The *getgrent*( ) function may fail if:

8978 [EINTR] A signal was caught during the operation.

8979 [EIO] An I/O error has occurred.

8980 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

8981 [ENFILE] The maximum allowable number of files is currently open in the system.

EXAMPLES

None.

APPLICATION USAGE

These functions are provided due to their historical usage. Applications should avoid dependencies on fields in the group database, whether the database is a single file, or where in the file system name space the database resides. Applications should use *getgrnam*() and *getgrgid*() whenever possible because it avoids these dependencies.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getgrgid(), *getgrnam*(), *getlogin*(), *getpwent*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<grp.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

Issue 6

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

9005 **NAME**

9006 endhostent, gethostent, sethostent — network host database functions

9007 **SYNOPSIS**

9008
```
#include <netdb.h>
```

9009
```
void endhostent(void);
```
9010
```
struct hostent *gethostent(void);
```
9011
```
void sethostent(int stayopen);
```

9012 **DESCRIPTION**

9013 These functions shall retrieve information about hosts. This information is considered to be |
9014 stored in a database that can be accessed sequentially or randomly. The implementation of this |
9015 database is unspecified. |

9016 **Note:** In many cases it is implemented by the Domain Name System, as documented in RFC 1034,
9017 RFC 1035, and RFC 1886.

9018 The *sethostent*() function shall open a connection to the database and set the next entry for
9019 retrieval to the first entry in the database. If the *stayopen* argument is non-zero, the connection
9020 shall not be closed by a call to *gethostent*(), *gethostbyname*(), or *gethostbyaddr*(), and the
9021 implementation may maintain an open file descriptor.

9022 The *gethostent*() function shall read the next entry in the database, opening and closing a
9023 connection to the database as necessary.

9024 Entries shall be returned in **hostent** structures. Refer to *gethostbyaddr*() for a definition of the |
9025 **hostent** structure.

9026 The *endhostent*() function shall close the connection to the database, releasing any open file
9027 descriptor.

9028 These functions need not be reentrant. A function that is not required to be reentrant is not
9029 required to be thread-safe.

9030 **RETURN VALUE**

9031 Upon successful completion, the *gethostent*() function shall return a pointer to a **hostent**
9032 structure if the requested entry was found, and a null pointer if the end of the database was
9033 reached or the requested entry was not found.

9034 **ERRORS**

9035 No errors are defined for *endhostent*(), *gethostent*(), and *sethostent*().

9036 **EXAMPLES**

9037 None.

9038 **APPLICATION USAGE**

9039 The *gethostent*() function may return pointers to static data, which may be overwritten by
9040 subsequent calls to any of these functions.

9041 **RATIONALE**

9042 None.

9043 **FUTURE DIRECTIONS**

9044 None.

9045 **SEE ALSO**

9046 *endservent*(), *gethostbyaddr*(), *gethostbyname*(), the Base Definitions volume of
9047 IEEE Std 1003.1-200x, **<netdb.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

9050 **NAME**

9051 endnetent, getnetbyaddr, getnetbyname, getnetent, setnetent — network database functions

9052 **SYNOPSIS**

9053
```
#include <netdb.h>
```

9054
```
void endnetent(void);
```
9055
```
struct netent *getnetbyaddr(uint32_t net, int type);
```
9056
```
struct netent *getnetbyname(const char *name);
```
9057
```
struct netent *getnetent(void);
```
9058
```
void setnetent(int stayopen);
```

9059 **DESCRIPTION**

9060 These functions shall retrieve information about networks. This information is considered to be |
9061 stored in a database that can be accessed sequentially or randomly. The implementation of this |
9062 database is unspecified. |

9063 The *setnetent*( ) function shall open and rewind the database. If the *stayopen* argument is non-
9064 zero, the connection to the *net* database shall not be closed after each call to *getnetent*( ) (either
9065 directly, or indirectly through one of the other *getnet**( ) functions), and the implementation may
9066 maintain an open file descriptor to the database.

9067 The *getnetent*( ) function shall read the next entry of the database, opening and closing a
9068 connection to the database as necessary.

9069 The *getnetbyaddr*( ) function shall search the database from the beginning, and find the first entry
9070 for which the address family specified by *type* matches the *n_addrtype* member and the network
9071 number *net* matches the *n_net* member, opening and closing a connection to the database as
9072 necessary. The *net* argument shall be the network number in host byte order.

9073 The *getnetbyname*( ) function shall search the database from the beginning and find the first entry
9074 for which the network name specified by *name* matches the *n_name* member, opening and
9075 closing a connection to the database as necessary.

9076 The *getnetbyaddr*( ), *getnetbyname*( ), and *getnetent*( ), functions shall each return a pointer to a
9077 **netent** structure, the members of which shall contain the fields of an entry in the network
9078 database.

9079 The *endnetent*( ) function shall close the database, releasing any open file descriptor.

9080 These functions need not be reentrant. A function that is not required to be reentrant is not
9081 required to be thread-safe.

9082 **RETURN VALUE**

9083 Upon successful completion, *getnetbyaddr*( ), *getnetbyname*( ), and *getnetent*( ), shall return a
9084 pointer to a **netent** structure if the requested entry was found, and a null pointer if the end of the
9085 database was reached or the requested entry was not found. Otherwise, a null pointer shall be
9086 returned.

9087 **ERRORS**

9088 No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *getnetbyaddr*(), *getnetbyname*(), and *getnetent*(), functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

9102 **NAME**

9103 endprotoent, getprotobyname, getprotobynumber, getprotoent, setprotoent — network protocol
9104 database functions

9105 **SYNOPSIS**

9106 ```
#include <netdb.h>
```

9107 ```
void endprotoent(void);
```
9108 ```
struct protoent *getprotobyname(const char *name);
```
9109 ```
struct protoent *getprotobynumber(int proto);
```
9110 ```
struct protoent *getprotoent(void);
```
9111 ```
void setprotoent(int stayopen);
```

9112 **DESCRIPTION**

9113 These functions shall retrieve information about protocols. This information is considered to be |
9114 stored in a database that can be accessed sequentially or randomly. The implementation of this |
9115 database is unspecified. |

9116 The *setprotoent*( ) function shall open a connection to the database, and set the next entry to the
9117 first entry. If the *stayopen* argument is non-zero, the connection to the network protocol database
9118 shall not be closed after each call to *getprotoent*( ) (either directly, or indirectly through one of the
9119 other *getproto**( ) functions), and the implementation may maintain an open file descriptor for
9120 the database.

9121 The *getprotobyname*( ) function shall search the database from the beginning and find the first
9122 entry for which the protocol name specified by *name* matches the *p_name* member, opening and
9123 closing a connection to the database as necessary.

9124 The *getprotobynumber*( ) function shall search the database from the beginning and find the first
9125 entry for which the protocol number specified by *proto* matches the *p_proto* member, opening
9126 and closing a connection to the database as necessary.

9127 The *getprotoent*( ) function shall read the next entry of the database, opening and closing a
9128 connection to the database as necessary.

9129 The *getprotobyname*( ), *getprotobynumber*( ), and *getprotoent*( ), functions shall each return a pointer
9130 to a **protoent** structure, the members of which shall contain the fields of an entry in the network
9131 protocol database.

9132 The *endprotoent*( ) function shall close the connection to the database, releasing any open file
9133 descriptor.

9134 These functions need not be reentrant. A function that is not required to be reentrant is not
9135 required to be thread-safe.

9136 **RETURN VALUE**

9137 Upon successful completion, *getprotobyname*( ), *getprotobynumber*( ), and *getprotoent*( ) return a
9138 pointer to a **protoent** structure if the requested entry was found, and a null pointer if the end of
9139 the database was reached or the requested entry was not found. Otherwise, a null pointer is
9140 returned.

9141 **ERRORS**

9142 No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *getprotobyname*(), *getprotobynumber*(), and *getprotoent*() functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

endpwent, getpwent, setpwent — user database functions

SYNOPSIS

XSI
```
#include <pwd.h>

void endpwent(void);
struct passwd *getpwent(void);
void setpwent(void);
```

DESCRIPTION

These functions shall retrieve information about users.

The *getpwent*( ) function shall return a pointer to a structure containing the broken-out fields of an entry in the user database. Each entry in the user database contains a **passwd** structure. When first called, *getpwent*( ) shall return a pointer to a **passwd** structure containing the first entry in the user database. Thereafter, it shall return a pointer to a **passwd** structure containing the next entry in the user database. Successive calls can be used to search the entire user database.

If an end-of-file or an error is encountered on reading, *getpwent*( ) shall return a null pointer.

An implementation that provides extended security controls may impose further implementation-defined restrictions on accessing the user database. In particular, the system may deny the existence of some or all of the user database entries associated with users other than the caller.

The *setpwent*( ) function effectively rewinds the user database to allow repeated searches.

The *endpwent*( ) function may be called to close the user database when processing is complete.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *getpwent*( ) function shall return a null pointer on end-of-file or error.

ERRORS

The *getpwent*( ), *setpwent*( ), and *endpwent*( ) functions may fail if:

[EIO] An I/O error has occurred.

In addition, *getpwent*( ) and *setpwent*( ) may fail if:

[EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

[ENFILE] The maximum allowable number of files is currently open in the system.

The return value may point to a static area which is overwritten by a subsequent call to *getpwuid*( ), *getpwnam*( ), or *getpwent*( ).

EXAMPLES

Searching the User Database

The following example uses the *getpwent*() function to get successive entries in the user database, returning a pointer to a **passwd** structure that contains information about each user. The call to *endpwent*() closes the user database and cleans up.

```
#include <pwd.h>
...
struct passwd *p;
...
while ((p = getpwent ()) != NULL) {
...
}

endpwent();
...
```

APPLICATION USAGE

These functions are provided due to their historical usage. Applications should avoid dependencies on fields in the password database, whether the database is a single file, or where in the file system name space the database resides. Applications should use *getpwuid*() whenever possible because it avoids these dependencies.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endgrent(), *getlogin*(), *getpwnam*(), *getpwuid*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<pwd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

Issue 6

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

9225 **NAME**

9226 endservent, getservbyname, getservbyport, getservent, setservent — network services database
9227 functions

9228 **SYNOPSIS**

9229 ```
#include <netdb.h>
```

9230 ```
void endservent(void);
```
9231 ```
struct servent *getservbyname(const char *name, const char *proto);
```
9232 ```
struct servent *getservbyport(int port, const char *proto);
```
9233 ```
struct servent *getservent(void);
```
9234 ```
void setservent(int stayopen);
```

9235 **DESCRIPTION**

9236 These functions shall retrieve information about network services. This information is
9237 considered to be stored in a database that can be accessed sequentially or randomly. The
9238 implementation of this database is unspecified.

9239 The *setservent*( ) function shall open a connection to the database, and set the next entry to the
9240 first entry. If the *stayopen* argument is non-zero, the *net* database shall not be closed after each
9241 call to the *getservent*( ) function (either directly, or indirectly through one of the other *getserv**( )
9242 functions), and the implementation may maintain an open file descriptor for the database.

9243 The *getservent*( ) function shall read the next entry of the database, opening and closing a
9244 connection to the database as necessary.

9245 The *getservbyname*( ) function shall search the database from the beginning and find the first
9246 entry for which the service name specified by *name* matches the *s_name* member and the protocol
9247 name specified by *proto* matches the *s_proto* member, opening and closing a connection to the
9248 database as necessary. If *proto* is a null pointer, any value of the *s_proto* member shall be
9249 matched.

9250 The *getservbyport*( ) function shall search the database from the beginning and find the first entry
9251 for which the port specified by *port* matches the *s_port* member and the protocol name specified
9252 by *proto* matches the *s_proto* member, opening and closing a connection to the database as
9253 necessary. If *proto* is a null pointer, any value of the *s_proto* member shall be matched. The *port*
9254 argument shall be in network byte order.

9255 The *getservbyname*( ), *getservbyport*( ), and *getservent*( ) functions shall each return a pointer to a
9256 **servent** structure, the members of which shall contain the fields of an entry in the network
9257 services database.

9258 The *endservent*( ) function shall close the database, releasing any open file descriptor.

9259 These functions need not be reentrant. A function that is not required to be reentrant is not
9260 required to be thread-safe.

9261 **RETURN VALUE**

9262 Upon successful completion, *getservbyname*( ), *getservbyport*( ), and *getservent*( ) return a pointer to
9263 a **servent** structure if the requested entry was found, and a null pointer if the end of the database
9264 was reached or the requested entry was not found. Otherwise, a null pointer is returned.

9265 **ERRORS**

9266 No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *port* argument of *getservbyport*() need not be compatible with the port values of all address families.

The *getservbyname*(), *getservbyport*(), and *getservent*() functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endhostent(), *endprotoent*(), *htonl*(), *inet_addr*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

endutxent, getutxent, getutxid, getutxline, pututxline, setutxent — user accounting database functions

SYNOPSIS

XSI

```
#include <utmpx.h>

void endutxent(void);
struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *id);
struct utmpx *getutxline(const struct utmpx *line);
struct utmpx *pututxline(const struct utmpx *utmpx);
void setutxent(void);
```

DESCRIPTION

These functions shall provide access to the user accounting database.

The *getutxent*( ) function shall read the next entry from the user accounting database. If the database is not already open, it shall open it. If it reaches the end of the database, it shall fail.

The *getutxid*( ) function shall search forward from the current point in the database. If the *ut_type* value of the **utmpx** structure pointed to by *id* is BOOT_TIME, OLD_TIME, or NEW_TIME, then it shall stop when it finds an entry with a matching *ut_type* value. If the *ut_type* value is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, or DEAD_PROCESS, then it shall stop when it finds an entry whose type is one of these four and whose *ut_id* member matches the *ut_id* member of the **utmpx** structure pointed to by *id*. If the end of the database is reached without a match, *getutxid*( ) shall fail.

The *getutxline*( ) function shall search forward from the current point in the database until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a *ut_line* value matching that in the **utmpx** structure pointed to by *line*. If the end of the database is reached without a match, *getutxline*( ) shall fail.

The *getutxid*( ) or *getutxline*( ) function may cache data. For this reason, to use *getutxline*( ) to search for multiple occurrences, the application shall zero out the static data after each success, or *getutxline*( ) may return a pointer to the same **utmpx** structure.

There is one exception to the rule about clearing the structure before further reads are done. The implicit read done by *pututxline*( ) (if it finds that it is not already at the correct place in the user accounting database) shall not modify the static structure returned by *getutxent*( ), *getutxid*( ), or *getutxline*( ), if the application has modified this structure and passed the pointer back to *pututxline*( ).

For all entries that match a request, the *ut_type* member indicates the type of the entry. Other members of the entry shall contain meaningful data based on the value of the *ut_type* member as follows:

ut_type Member	Other Members with Meaningful Data
EMPTY	No others
BOOT_TIME	*ut_tv*
OLD_TIME	*ut_tv*
NEW_TIME	*ut_tv*
USER_PROCESS	*ut_id*, *ut_user* (login name of the user), *ut_line*, *ut_pid*, *ut_tv*
INIT_PROCESS	*ut_id*, *ut_pid*, *ut_tv*
LOGIN_PROCESS	*ut_id*, *ut_user* (implementation-defined name of the login process), *ut_pid*, *ut_tv*
DEAD_PROCESS	*ut_id*, *ut_pid*, *ut_tv*

An implementation that provides extended security controls may impose implementation-defined restrictions on accessing the user accounting database. In particular, the system may deny the existence of some or all of the user accounting database entries associated with users other than the caller.

If the process has appropriate privileges, the *pututxline*( ) function shall write out the structure into the user accounting database. It shall use *getutxid*( ) to search for a record that satisfies the request. If this search succeeds, then the entry shall be replaced. Otherwise, a new entry shall be made at the end of the user accounting database.

The *endutxent*( ) function shall close the user accounting database.

The *setutxent*( ) function shall reset the input to the beginning of the database. This should be done before each search for a new entry if it is desired that the entire database be examined.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *getutxent*( ), *getutxid*( ), and *getutxline*( ) shall return a pointer to a **utmpx** structure containing a copy of the requested entry in the user accounting database. Otherwise, a null pointer shall be returned.

The return value may point to a static area which is overwritten by a subsequent call to *getutxid*( ) or *getutxline*( ).

Upon successful completion, *pututxline*( ) shall return a pointer to a **utmpx** structure containing a copy of the entry added to the user accounting database. Otherwise, a null pointer shall be returned.

The *endutxent*( ) and *setutxent*( ) functions shall not return a value.

ERRORS

No errors are defined for the *endutxent*( ), *getutxent*( ), *getutxid*( ), *getutxline*( ), and *setutxent*( ) functions.

The *pututxline*( ) function may fail if:

[EPERM] The process does not have appropriate privileges.

9360 **EXAMPLES**

9361 None.

9362 **APPLICATION USAGE**

9363 The sizes of the arrays in the structure can be found using the *sizeof* operator.

9364 **RATIONALE**

9365 None.

9366 **FUTURE DIRECTIONS**

9367 None.

9368 **SEE ALSO**

9369 The Base Definitions volume of IEEE Std 1003.1-200x, **<utmpx.h>**

9370 **CHANGE HISTORY**

9371 First released in Issue 4, Version 2.

9372 **Issue 5**

9373 Moved from X/OPEN UNIX extension to BASE.

9374 Normative text previously in the APPLICATION USAGE section is moved to the
9375 DESCRIPTION.

9376 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

9377 **Issue 6**

9378 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

9379 **NAME**

9380 environ — array of character pointers to the environment strings

9381 **SYNOPSIS**

9382
```
extern char **environ;
```

9383 **DESCRIPTION**

9384 Refer to the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables
9385 and *exec*.

9386 **NAME**

9387 erand48 — generate uniformly distributed pseudo-random numbers

9388 **SYNOPSIS**

9389 XSI
```
#include <stdlib.h>
```

9390
```
double erand48(unsigned short xsubi[3]);
```

9391

9392 **DESCRIPTION**

9393 Refer to *drand48*( ).

9394 **NAME**

9395 erf, erff, erfl — error functions

9396 **SYNOPSIS**

9397
```
#include <math.h>
```

9398
```
double erf(double x);
```
9399
```
float erff(float x);
```
9400
```
long double erfl(long double x);
```

9401 **DESCRIPTION**

9402 CX The functionality described on this reference page is aligned with the ISO C standard. Any
9403 conflict between the requirements described here and the ISO C standard is unintentional. This
9404 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

9405 These functions shall compute the error function of their argument *x*, defined as:

9406 $$\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$$

9407 An application wishing to check for error situations should set *errno* to zero and call
9408 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
9409 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
9410 zero, an error has occurred.

9411 **RETURN VALUE**

9412 Upon successful completion, these functions shall return the value of the error function.

9413 MX If *x* is NaN, a NaN shall be returned.

9414 If *x* is ±0, ±0 shall be returned.

9415 If *x* is ±Inf, ±1 shall be returned.

9416 If *x* is subnormal, a range error may occur, and 2 * *x* /*sqrt*(π) should be returned.

9417 **ERRORS**

9418 These functions may fail if:

9419 MX

Range Error	The result underflows.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

9424 **EXAMPLES**

9425 None.

9426 **APPLICATION USAGE**

9427 Underflow occurs when |*x*| < DBL_MIN * (*sqrt*(π) / 2).

9428 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
9429 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

9430 **RATIONALE**

9431 None.

9432 **FUTURE DIRECTIONS**

9433 None.

9434 **SEE ALSO**

9435 *erfc*( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, |
9436 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

9437 **CHANGE HISTORY**

9438 First released in Issue 1. Derived from Issue 1 of the SVID.

9439 **Issue 5**

9440 The DESCRIPTION is updated to indicate how an application should check for an error. This
9441 text was previously published in the APPLICATION USAGE section.

9442 **Issue 6**

9443 The *erf*( ) function is no longer marked as an extension.

9444 The *erfc*( ) function is now split out onto its own reference page.

9445 The *erff*( ) and *erfl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

9446 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
9447 revised to align with the ISO / IEC 9899: 1999 standard.

9448 IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are
9449 marked.

NAME

erfc, erfcf, erfcl — complementary error functions

SYNOPSIS

```
#include <math.h>

double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complementary error function 1.0 – *erf*(*x*).

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the value of the complementary error function.

If the correct value would cause underflow and is not representable, a range error may occur and MX either 0.0 (if representable), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, +1 shall be returned.

If *x* is –Inf, +2 shall be returned.

If *x* is +Inf, +0 shall be returned.

If the correct value would cause underflow and is representable, a range error may occur and the correct value shall be returned.

ERRORS

These functions may fail if:

Range Error — The result underflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

The *erfc*( ) function is provided because of the extreme loss of relative accuracy if *erf*(*x*) is called for large *x* and the result subtracted from 1.0.

Note for IEEE Std 754-1985 **double**, 26.55 < *x* implies *erfc*(*x*) has underflowed.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

9492 **RATIONALE**

9493 None.

9494 **FUTURE DIRECTIONS**

9495 None.

9496 **SEE ALSO**

9497 *erf*( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, |
9498 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

9499 **CHANGE HISTORY**

9500 First released in Issue 1. Derived from Issue 1 of the SVID.

9501 **Issue 5**

9502 The DESCRIPTION is updated to indicate how an application should check for an error. This
9503 text was previously published in the APPLICATION USAGE section.

9504 **Issue 6**

9505 The *erfc*( ) function is no longer marked as an extension.

9506 These functions are split out from the *erf*( ) reference page.

9507 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
9508 revised to align with the ISO / IEC 9899: 1999 standard.

9509 IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are
9510 marked.

9511 **NAME**

9512 erff, erfl — error functions

9513 **SYNOPSIS**

9514 `#include <math.h>`

9515 `float erff(float x);`

9516 `long double erfl(long double x);`

9517 **DESCRIPTION**

9518 Refer to *erf*( ).

9519 **NAME**

9520 errno — error return value

9521 **SYNOPSIS**

9522 `#include <errno.h>`

9523 **DESCRIPTION**

9524 The lvalue *errno* is used by many functions to return error values. |

9525 Many functions provide an error number in *errno*. It has type **int** and is defined in **<errno.h>**. |
9526 The value of *errno* shall be defined only after a call to a function for which it is explicitly stated to
9527 be set and until it is changed by the next function call or if the application assigns it a value. The |
9528 value of *errno* should only be examined when it is indicated to be valid by a function's return |
9529 value. Applications shall obtain the definition of *errno* by the inclusion of **<errno.h>**. No |
9530 function in this volume of IEEE Std 1003.1-200x shall set *errno* to 0.

9531 It is unspecified whether *errno* is a macro or an identifier declared with external linkage. If a
9532 macro definition is suppressed in order to access an actual object, or a program defines an
9533 identifier with the name *errno*, the behavior is undefined.

9534 The symbolic values stored in *errno* are documented in the ERRORS sections on all relevant
9535 pages.

9536 **RETURN VALUE**

9537 None.

9538 **ERRORS**

9539 None.

9540 **EXAMPLES**

9541 None.

9542 **APPLICATION USAGE**

9543 Previously both POSIX and X/Open documents were more restrictive than the ISO C standard
9544 in that they required *errno* to be defined as an external variable, whereas the ISO C standard
9545 required only that *errno* be defined as a modifiable lvalue with type **int**. |

9546 A program that uses *errno* for error checking should set it to 0 before a function call, then inspect
9547 it before a subsequent function call.

9548 **RATIONALE**

9549 None.

9550 **FUTURE DIRECTIONS**

9551 None.

9552 **SEE ALSO**

9553 Section 2.3, the Base Definitions volume of IEEE Std 1003.1-200x, **<errno.h>**

9554 **CHANGE HISTORY**

9555 First released in Issue 1. Derived from Issue 1 of the SVID.

9556 **Issue 5**

9557 The following sentence is deleted from the DESCRIPTION: ''The value of *errno* is 0 at program
9558 start-up, but is never set to 0 by any XSI function''. The DESCRIPTION also no longer states that
9559 conforming implementations may support the declaration:

9560 `extern int errno;`

Issue 6

Obsolescent text regarding defining *errno* as:

```
extern int errno
```

is removed.

Text regarding no function setting *errno* to zero to indicate an error is changed to no function shall set *errno* to zero. This is for alignment with the ISO / IEC 9899: 1999 standard.

9567 **NAME**

9568 environ, execl, execv, execle, execve, execlp, execvp — execute a file

9569 **SYNOPSIS**

9570
```
#include <unistd.h>

extern char **environ;
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ... /*,
    (char *)0, char *const envp[]*/);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execvp(const char *file, char *const argv[]);
```

9579 **DESCRIPTION**

9580 The *exec* family of functions shall replace the current process image with a new process image. |
9581 The new image shall be constructed from a regular, executable file called the *new process image* |
9582 *file*. There shall be no return from a successful *exec*, because the calling process image is overlaid
9583 by the new process image.

9584 When a C-language program is executed as a result of this call, it shall be entered as a C-
9585 language function call as follows:

9586
```
int main (int argc, char *argv[]);
```

9587 where *argc* is the argument count and *argv* is an array of character pointers to the arguments
9588 themselves. In addition, the following variable:

9589
```
extern char **environ;
```

9590 is initialized as a pointer to an array of character pointers to the environment strings. The *argv*
9591 and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv*
9592 array is not counted in *argc*.

9593 THR Conforming multi-threaded applications shall not use the *environ* variable to access or modify
9594 any environment variable while any other thread is concurrently modifying any environment
9595 variable. A call to any function dependent on any environment variable shall be considered a use
9596 of the *environ* variable to access that environment variable.

9597 The arguments specified by a program with one of the *exec* functions shall be passed on to the
9598 new process image in the corresponding *main*( ) arguments.

9599 The argument *path* points to a pathname that identifies the new process image file. |

9600 The argument *file* is used to construct a pathname that identifies the new process image file. If |
9601 the *file* argument contains a slash character, the *file* argument shall be used as the pathname for |
9602 this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed |
9603 as the environment variable *PATH* (see the Base Definitions volume of IEEE Std 1003.1-200x,
9604 Chapter 8, Environment Variables). If this environment variable is not present, the results of the
9605 search are implementation-defined.

9606 If the process image file is not a valid executable object, and the system does not recognize it as
9607 something that cannot be executed (and thus returns [EINVAL]), *execlp*( ) and *execvp*( ) shall use
9608 the contents of that file as standard input to a command interpreter conforming to *system*( ). In
9609 this case, the command interpreter becomes the new process image.

9610 The arguments represented by *arg0*,... are pointers to null-terminated character strings. These
9611 strings shall constitute the argument list available to the new process image. The list is |

terminated by a null pointer. The argument *arg0* should point to a filename that is associated with the process being started by one of the *exec* functions.

The argument *argv* is an array of character pointers to null-terminated strings. The application shall ensure that the last member of this array is a null pointer. These strings shall constitute the argument list available to the new process image. The value in *argv*[0] should point to a filename that is associated with the process being started by one of the *exec* functions.

The argument *envp* is an array of character pointers to null-terminated strings. These strings shall constitute the environment for the new process image. The *envp* array is terminated by a null pointer.

For those forms not containing an *envp* pointer (*execl*( ), *execv*( ), *execlp*( ), and *execvp*( )), the environment for the new process image shall be taken from the external variable *environ* in the calling process.

The number of bytes available for the new process' combined argument and environment lists is {ARG_MAX}. It is implementation-defined whether null terminators, pointers, and/or any alignment bytes are included in this total.

File descriptors open in the calling process image shall remain open in the new process image, except for those whose close-on-exec flag FD_CLOEXEC is set. For those file descriptors that remain open, all attributes of the open file description remain unchanged. For any file descriptor that is closed for this reason, file locks are removed as a result of the close as described in *close*( ). Locks that are not removed by closing of file descriptors remain unchanged.

Directory streams open in the calling process image shall be closed in the new process image.

The state of the floating-point environment in the new process image shall be set to the default.

XSI The state of conversion descriptors and message catalog descriptors in the new process image is undefined. For the new process image, the equivalent of:

```
setlocale(LC_ALL, "C")
```

shall be executed at start-up.

Signals set to the default action (SIG_DFL) in the calling process image shall be set to the default action in the new process image. Except for SIGCHLD, signals set to be ignored (SIG_IGN) by the calling process image shall be set to be ignored by the new process image. Signals set to be caught by the calling process image shall be set to the default action in the new process image (see **<signal.h>**). If the SIGCHLD signal is set to be ignored by the calling process image, it is unspecified whether the SIGCHLD signal is set to be ignored or to the default action in the new process image. XSI After a successful call to any of the *exec* functions, alternate signal stacks are not preserved and the SA_ONSTACK flag shall be cleared for all signals.

After a successful call to any of the *exec* functions, any functions previously registered by *atexit*( ) are no longer registered.

XSI If the ST_NOSUID bit is set for the file system containing the new process image file, then the effective user ID, effective group ID, saved set-user-ID, and saved set-group-ID are unchanged in the new process image. Otherwise, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the new process image shall be set to the user ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image shall be set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image shall remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image shall be saved (as the saved set-user-ID and the saved set-group-ID) for use by *setuid*( ).

XSI Any shared memory segments attached to the calling process image shall not be attached to the new process image.

SEM Any named semaphores open in the calling process shall be closed as if by appropriate calls to *sem_close*( ).

TYM Any blocks of typed memory that were mapped in the calling process are unmapped, as if *munmap*( ) was implicitly called to unmap them.

ML Memory locks established by the calling process via calls to *mlockall*( ) or *mlock*( ) shall be removed. If locked pages in the address space of the calling process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes shall be unaffected by the call by this process to the *exec* function. If the *exec* function fails, the effect on memory locks is unspecified.

MF|SHM Memory mappings created in the process are unmapped before the address space is rebuilt for the new process image.

PS For the SCHED_FIFO and SCHED_RR scheduling policies, the policy and priority settings shall not be changed by a call to an *exec* function. For other scheduling policies, the policy and priority settings on *exec* are implementation-defined.

TMR Per-process timers created by the calling process shall be deleted before replacing the current process image with the new process image.

MSG All open message queue descriptors in the calling process shall be closed, as described in *mq_close*( ).

AIO Any outstanding asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled shall complete as if the *exec* function had not yet occurred, but any associated signal notifications shall be suppressed. It is unspecified whether the *exec* function itself blocks awaiting such I/O completion. In no event, however, shall the new process image created by the *exec* function be affected by the presence of outstanding asynchronous I/O operations at the time the *exec* function is called. Whether any I/O is canceled, and which I/O may be canceled upon *exec*, is implementation-defined.

CPT The new process image shall inherit the CPU-time clock of the calling process image. This inheritance means that the process CPU-time clock of the process being *exec*ed shall not be reinitialized or altered as a result of the *exec* function other than to reflect the time spent by the process executing the *exec* function itself.

TCT The initial value of the CPU-time clock of the initial thread of the new process image shall be set to zero.

TRC If the calling process is being traced, the new process image shall continue to be traced into the same trace stream as the original process image, but the new process image shall not inherit the mapping of trace event names to trace event type identifiers that was defined by calls to the *posix_trace_eventid_open*( ) or the *posix_trace_trid_eventid_open*( ) functions in the calling process image.

If the calling process is a trace controller process, any trace streams that were created by the calling process shall be shut down as described in the *posix_trace_shutdown*( ) function.

The new process shall inherit at least the following attributes from the calling process image:

XSI
- Nice value (see *nice*( ))

XSI
- *semadj* values (see *semop*( ))

- Process ID
- Parent process ID
- Process group ID
- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs
- Time left until an alarm clock signal (see *alarm*( ))
- Current working directory
- Root directory
- File mode creation mask (see *umask*( ))

XSI
- File size limit (see *ulimit*( ))

- Process signal mask (see *sigprocmask*( ))
- Pending signal (see *sigpending*( ))
- *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see *times*( ))

XSI
- Resource limits

XSI
- Controlling terminal

XSI
- Interval timers

All other process attributes defined in this volume of IEEE Std 1003.1-200x shall be the same in the new and old process images. The inheritance of process attributes not defined by this volume of IEEE Std 1003.1-200x is implementation-defined.

A call to any *exec* function from a process with more than one thread shall result in all threads being terminated and the new executable image being loaded and executed. No destructor functions shall be called.

Upon successful completion, the *exec* functions shall mark for update the *st_atime* field of the file. If an *exec* function failed but was able to locate the *process image file*, whether the *st_atime* field is marked for update is unspecified. Should the *exec* function succeed, the process image file shall be considered to have been opened with *open*( ). The corresponding *close*( ) shall be considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the *exec* functions, *posix_spawn*( ), or *posix_spawnp*( ). The *argv*[ ] and *envp*[ ] arrays of pointers and the strings to which those arrays point shall not be modified by a call to one of the *exec* functions, except as a consequence of replacing the process image.

XSI The saved resource limits in the new process image are set to be a copy of the process' corresponding hard and soft limits.

RETURN VALUE

If one of the *exec* functions returns to the calling process image, an error has occurred; the return value shall be –1, and *errno* shall be set to indicate the error.

9738 **ERRORS**

9739 The *exec* functions shall fail if:

9740 [E2BIG] The number of bytes used by the new process image's argument list and
9741 environment list is greater than the system-imposed limit of {ARG_MAX}
9742 bytes.

9743 [EACCES] Search permission is denied for a directory listed in the new process image
9744 file's path prefix, or the new process image file denies execution permission,
9745 or the new process image file is not a regular file and the implementation does
9746 not support execution of files of its type.

9747 [EINVAL] The new process image file has the appropriate permission and has a
9748 recognized executable binary format, but the system does not support
9749 execution of a file with this format.

9750 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* or *file*
9751 argument.

9752 [ENAMETOOLONG]
9753 The length of the *path* or *file* arguments exceeds {PATH_MAX} or a pathname |
9754 component is longer than {NAME_MAX}. |

9755 [ENOENT] A component of *path* or *file* does not name an existing file or *path* or *file* is an
9756 empty string.

9757 [ENOTDIR] A component of the new process image file's path prefix is not a directory.

9758 The *exec* functions, except for *execlp*( ) and *execvp*( ), shall fail if:

9759 [ENOEXEC] The new process image file has the appropriate access permission but has an
9760 unrecognized format.

9761 The *exec* functions may fail if:

9762 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
9763 resolution of the *path* or *file* argument.

9764 [ENAMETOOLONG]
9765 As a result of encountering a symbolic link in resolution of the *path* argument, |
9766 the length of the substituted pathname string exceeded {PATH_MAX}. |

9767 [ENOMEM] The new process image requires more memory than is allowed by the
9768 hardware or system-imposed memory management constraints.

9769 [ETXTBSY] The new process image file is a pure procedure (shared text) file that is
9770 currently open for writing by some process.

9771 **EXAMPLES**

9772 **Using execl( )**

9773 The following example executes the *ls* command, specifying the pathname of the executable |
9774 (**/bin/ls**) and using arguments supplied directly to the command to produce single-column
9775 output.

```
9776 #include <unistd.h>
9777 int ret;
9778 ...
9779 ret = execl ("/bin/ls", "ls", "-1", (char *)0);
```

Using execle()

The following example is similar to **Using execl()** (on page 758). In addition, it specifies the environment for the new process image using the *env* argument.

```
#include <unistd.h>

int ret;
char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
...
ret = execle ("/bin/ls", "ls", "-l", (char *)0, env);
```

Using execlp()

The following example searches for the location of the *ls* command among the directories specified by the *PATH* environment variable.

```
#include <unistd.h>

int ret;
...
ret = execlp ("ls", "ls", "-l", (char *)0);
```

Using execv()

The following example passes arguments to the *ls* command in the *cmd* array.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
...
ret = execv ("/bin/ls", cmd);
```

Using execve()

The following example passes arguments to the *ls* command in the *cmd* array, and specifies the environment for the new process image using the *env* argument.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
...
ret = execve ("/bin/ls", cmd, env);
```

Using execvp()

The following example searches for the location of the *ls* command among the directories specified by the *PATH* environment variable, and passes arguments to the *ls* command in the *cmd* array.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
...
```

```
ret = execvp ("ls", cmd);
```

APPLICATION USAGE

As the state of conversion descriptors and message catalog descriptors in the new process image is undefined, conforming applications should not rely on their use and should close them prior to calling one of the *exec* functions.

Applications that require other than the default POSIX locale should call *setlocale*() with the appropriate parameters to establish the locale of the new process.

The *environ* array should not be accessed directly by the application.

RATIONALE

Early proposals required that the value of *argc* passed to *main*() be ''one or greater''. This was driven by the same requirement in drafts of the ISO C standard. In fact, historical implementations have passed a value of zero when no arguments are supplied to the caller of the *exec* functions. This requirement was removed from the ISO C standard and subsequently removed from this volume of IEEE Std 1003.1-200x as well. The wording, in particular the use of the word *should*, requires a Strictly Conforming POSIX Application to pass at least one argument to the *exec* function, thus guaranteeing that *argc* be one or greater when invoked by such an application. In fact, this is good practice, since many existing applications reference *argv*[0] without first checking the value of *argc*.

The requirement on a Strictly Conforming POSIX Application also states that the value passed as the first argument be a filename associated with the process being started. Although some existing applications pass a pathname rather than a filename in some circumstances, a filename is more generally useful, since the common usage of *argv*[0] is in printing diagnostics. In some cases the filename passed is not the actual filename of the file; for example, many implementations of the *login* utility use a convention of prefixing a hyphen ('-') to the actual filename, which indicates to the command interpreter being invoked that it is a ''login shell''.

Some implementations can *exec* shell scripts.

One common historical implementation is that the *execl*(), *execv*(), *execle*(), and *execve*() functions return an [ENOEXEC] error for any file not recognizable as executable, including a shell script. When the *execlp*() and *execvp*() functions encounter such a file, they assume the file to be a shell script and invoke a known command interpreter to interpret such files. These implementations of *execvp*() and *execlp*() only give the [ENOEXEC] error in the rare case of a problem with the command interpreter's executable file. Because of these implementations, the [ENOEXEC] error is not mentioned for *execlp*() or *execvp*(), although implementations can still give it.

Another way that some historical implementations handle shell scripts is by recognizing the first two bytes of the file as the character string "#!" and using the remainder of the first line of the file as the name of the command interpreter to execute.

Some implementations provide a third argument to *main*() called *envp*. This is defined as a pointer to the environment. The ISO C standard specifies invoking *main*() with two arguments, so implementations must support applications written this way. Since this volume of IEEE Std 1003.1-200x defines the global variable *environ*, which is also provided by historical implementations and can be used anywhere that *envp* could be used, there is no functional need for the *envp* argument. Applications should use the *getenv*() function rather than accessing the environment directly via either *envp* or *environ*. Implementations are required to support the two-argument calling sequence, but this does not prohibit an implementation from supporting *envp* as an optional third argument.

This volume of IEEE Std 1003.1-200x specifies that signals set to SIG_IGN remain set to SIG_IGN, and that the process signal mask be unchanged across an *exec*. This is consistent with historical implementations, and it permits some useful functionality, such as the *nohup* command. However, it should be noted that many existing applications wrongly assume that they start with certain signals set to the default action and/or unblocked. In particular, applications written with a simpler signal model that does not include blocking of signals, such as the one in the ISO C standard, may not behave properly if invoked with some signals blocked. Therefore, it is best not to block or ignore signals across *exec*s without explicit reason to do so, and especially not to block signals across *exec*s of arbitrary (not closely co-operating) programs.

The *exec* functions always save the value of the effective user ID and effective group ID of the process at the completion of the *exec*, whether or not the set-user-ID or the set-group-ID bit of the process image file is set.

The statement about *argv*[ ] and *envp*[ ] being constants is included to make explicit to future writers of language bindings that these objects are completely constant. Due to a limitation of the ISO C standard, it is not possible to state that idea in standard C. Specifying two levels of *const–qualification* for the *argv*[ ] and *envp*[ ] parameters for the *exec* functions may seem to be the natural choice, given that these functions do not modify either the array of pointers or the characters to which the function points, but this would disallow existing correct code. Instead, only the array of pointers is noted as constant. The table of assignment compatibility for *dst=src*, derived from the ISO C standard summarizes the compatibility:

dst:	**char *[ ]**	**const char *[ ]**	**char *const[ ]**	**const char *const[ ]**
src:				
char *[ ]	VALID	—	VALID	—
const char *[ ]	—	VALID	—	VALID
char * const [ ]	—	—	VALID	—
const char *const[ ]	—	—	—	VALID

Since all existing code has a source type matching the first row, the column that gives the most valid combinations is the third column. The only other possibility is the fourth column, but using it would require a cast on the *argv* or *envp* arguments. It is unfortunate that the fourth column cannot be used, because the declaration a non-expert would naturally use would be that in the second row.

The ISO C standard and this volume of IEEE Std 1003.1-200x do not conflict on the use of *environ*, but some historical implementations of *environ* may cause a conflict. As long as *environ* is treated in the same way as an entry point (for example, *fork*( )), it conforms to both standards. A library can contain *fork*( ), but if there is a user-provided *fork*( ), that *fork*( ) is given precedence and no problem ensues. The situation is similar for *environ*: the definition in this volume of IEEE Std 1003.1-200x is to be used if there is no user-provided *environ* to take precedence. At least three implementations are known to exist that solve this problem.

[E2BIG] The limit {ARG_MAX} applies not just to the size of the argument list, but to the sum of that and the size of the environment list.

[EFAULT] Some historical systems return [EFAULT] rather than [ENOEXEC] when the new process image file is corrupted. They are non-conforming.

[EINVAL] This error condition was added to IEEE Std 1003.1-200x to allow an implementation to detect executable files generated for different architectures, and indicate this situation to the application. Historical implementations of shells, *execvp*( ), and *execlp*( ) that encounter an [ENOEXEC] error will execute a shell on the assumption that the file is a shell script. This will not produce the desired effect when the file is a valid executable for a different

architecture. An implementation may now choose to avoid this problem by returning [EINVAL] when a valid executable for a different architecture is encountered. Some historical implementations return [EINVAL] to indicate that the *path* argument contains a character with the high order bit set. The standard developers chose to deviate from historical practice for the following reasons:

1. The new utilization of [EINVAL] will provide some measure of utility to the user community.
2. Historical use of [EINVAL] is not acceptable in an internationalized operating environment.

[ENAMETOOLONG]
Since the file pathname may be constructed by taking elements in the *PATH* variable and putting them together with the filename, the [ENAMETOOLONG] error condition could also be reached this way.

[ETXTBSY] System V returns this error when the executable file is currently open for writing by some process. This volume of IEEE Std 1003.1-200x neither requires nor prohibits this behavior.

Other systems (such as System V) may return [EINTR] from *exec*. This is not addressed by this volume of IEEE Std 1003.1-200x, but implementations may have a window between the call to *exec* and the time that a signal could cause one of the *exec* calls to return with [EINTR].

An explicit statement regarding the floating-point environment (as defined in the **<fenv.h>** header) was added to make it clear that the floating-point environment is set to its default when a call to one of the *exec* functions succeeds. The requirements for inheritance or setting to the default for other process and thread start-up functions is covered by more generic statements in their descriptions and can be summarized as follows:

posix_spawn( )	**Set to default.**
fork( )	Inherit.
pthread_create( )	Inherit.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *atexit*( ), *chmod*( ), *close*( ), *exit*( ), *fcntl*( ), *fork*( ), *fstatvfs*( ), *getenv*( ), *getitimer*( ), *getrlimit*( ), *mmap*( ), *nice*( ), *posix_spawn*( ), *posix_trace_eventid_open*( ), *posix_trace_shutdown*( ), *posix_trace_trid_eventid_open*( ), *putenv*( ), *semop*( ), *setlocale*( ), *shmat*( ), *sigaction*( ), *sigaltstack*( ), *sigpending*( ), *sigprocmask*( ), *system*( ), *times*( ), *ulimit*( ), *umask*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, behavior is defined for when the process image file is not a valid executable.
- In this issue, _POSIX_SAVED_IDS is mandated, thus the effective user ID and effective group ID of the new process image shall be saved (as the saved set-user-ID and the saved set-group-ID) for use by the *setuid*( ) function.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.
- The [ETXTBSY] optional error condition is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [EINVAL] mandatory error condition is added.
- The [ELOOP] optional error condition is added.

The description of CPU-time clock semantics is added for alignment with IEEE Std 1003.1d-1999.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for typed memory.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #132 is applied.

The DESCRIPTION is updated to make it explicit that the floating-point environment in the new process image is set to the default.

NAME

exit, _Exit, _exit — terminate a process

SYNOPSIS

```
#include <stdlib.h>

void exit(int status);
void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

DESCRIPTION

CX The functionality described on this reference page for the *exit*( ) and *_Exit*( ) functions is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard are unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The value of *status* may be 0, EXIT_SUCCESS, EXIT_FAILURE, or any other value, though only the least significant 8 bits (that is, *status* & 0377) shall be available to a waiting parent process.

The *exit*( ) function shall first call all functions registered by *atexit*( ), in the reverse order of their registration, except that a function is called after any previously registered functions that had already been called at the time it was registered. Each function is called as many times as it was registered. If, during the call to any such function, a call to the *longjmp*( ) function is made that would terminate the call to the registered function, the behavior is undefined.

If a function registered by a call to *atexit*( ) fails to return, the remaining registered functions shall not be called and the rest of the *exit*( ) processing shall not be completed. If *exit*( ) is called more than once, the behavior is undefined.

The *exit*( ) function shall then flush all open streams with unwritten buffered data, close all open streams, and remove all files created by *tmpfile*( ). Finally, control shall be terminated with the consequences described below.

CX The *_Exit*( ) and *_exit*( ) functions shall be functionally equivalent.

CX The *_Exit*( ) and *_exit*( ) functions shall not call functions registered with *atexit*( ) nor any registered signal handlers. Whether open streams are flushed or closed, or temporary files are removed is implementation-defined. Finally, the calling process is terminated with the consequences described below.

CX These functions shall terminate the calling process with the following consequences:

Note: These consequences are all extensions to the ISO C standard and are not further CX shaded. However, XSI extensions are shaded.

XSI - All of the file descriptors, directory streams, conversion descriptors, and message catalog descriptors open in the calling process shall be closed.

XSI - If the parent process of the calling process is executing a *wait*( ) or *waitpid*( ), and has neither set its SA_NOCLDWAIT flag nor set SIGCHLD to SIG_IGN, it shall be notified of the calling process' termination and the low-order eight bits (that is, bits 0377) of *status* are made available to it. If the parent is not waiting, the child's status shall be made available to it when the parent subsequently executes *wait*( ) or *waitpid*( ).

XSI The semantics of the *waitid*( ) function shall be equivalent to *wait*( ).

XSI - If the parent process of the calling process is not executing a *wait*( ) or *waitpid*( ), and has neither set its SA_NOCLDWAIT flag nor set SIGCHLD to SIG_IGN, the calling process shall be transformed into a *zombie process*. A *zombie process* is an inactive process and it shall be

deleted at some later time when its parent process executes *wait*( ) or *waitpid*( ).

XSI The semantics of the *waitid*( ) function shall be equivalent to *wait*( ).

- Termination of a process does not directly terminate its children. The sending of a SIGHUP signal as described below indirectly terminates children in some circumstances.
- Either:

  If the implementation supports the SIGCHLD signal, a SIGCHLD shall be sent to the parent process.

  Or:

  XSI If the parent process has set its SA_NOCLDWAIT flag, or set SIGCHLD to SIG_IGN, the status shall be discarded, and the lifetime of the calling process shall end immediately. If SA_NOCLDWAIT is set, it is implementation-defined whether a SIGCHLD signal is sent to the parent process.
- The parent process ID of all of the calling process' existing child processes and zombie processes shall be set to the process ID of an implementation-defined system process. That is, these processes shall be inherited by a special system process.
- XSI Each attached shared-memory segment is detached and the value of *shm_nattch* (see *shmget*( )) in the data structure associated with its shared memory ID shall be decremented by 1.
- XSI For each semaphore for which the calling process has set a *semadj* value (see *semop*( )), that value shall be added to the *semval* of the specified semaphore.
- If the process is a controlling process, the SIGHUP signal shall be sent to each process in the foreground process group of the controlling terminal belonging to the calling process.
- If the process is a controlling process, the controlling terminal associated with the session shall be disassociated from the session, allowing it to be acquired by a new controlling process.
- If the exit of the process causes a process group to become orphaned, and if any member of the newly-orphaned process group is stopped, then a SIGHUP signal followed by a SIGCONT signal shall be sent to each process in the newly-orphaned process group.
- SEM All open named semaphores in the calling process shall be closed as if by appropriate calls to *sem_close*( ).
- ML Any memory locks established by the process via calls to *mlockall*( ) or *mlock*( ) shall be removed. If locked pages in the address space of the calling process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes shall be unaffected by the call by this process to *_Exit*( ) or *_exit*( ).
- MF|SHM Memory mappings created in the process shall be unmapped before the process is destroyed.
- TYM Any blocks of typed memory that were mapped in the calling process shall be unmapped, as if *munmap*( ) was implicitly called to unmap them.
- MSG All open message queue descriptors in the calling process shall be closed as if by appropriate calls to *mq_close*( ).
- AIO Any outstanding cancelable asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled shall complete as if the *_Exit*( ) or *_exit*( ) operation had not yet occurred, but any associated signal notifications shall be suppressed.

The *_Exit*( ) or *_exit*( ) operation may block awaiting such I/O completion. Whether any I/O is canceled, and which I/O may be canceled upon *_Exit*( ) or *_exit*( ), is implementation-defined.

- Threads terminated by a call to *_Exit*( ) or *_exit*( ) shall not invoke their cancelation cleanup handlers or per-thread data destructors.

TRC
- If the calling process is a trace controller process, any trace streams that were created by the calling process shall be shut down as described by the *posix_trace_shutdown*( ) function, and any process' mapping of trace event names to trace event type identifiers built for these trace streams may be deallocated.

RETURN VALUE

These functions do not return.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Normally applications should use *exit*( ) rather than *_Exit*( ) or *_exit*( ).

RATIONALE

Process Termination

Early proposals drew a distinction between normal and abnormal process termination. Abnormal termination was caused only by certain signals and resulted in implementation-defined ''actions'', as discussed below. Subsequent proposals distinguished three types of termination: *normal termination* (as in the current specification), *simple abnormal termination*, and *abnormal termination with actions*. Again the distinction between the two types of abnormal termination was that they were caused by different signals and that implementation-defined actions would result in the latter case. Given that these actions were completely implementation-defined, the early proposals were only saying when the actions could occur and how their occurrence could be detected, but not what they were. This was of little or no use to conforming applications, and thus the distinction is not made in this volume of IEEE Std 1003.1-200x.

The implementation-defined actions usually include, in most historical implementations, the creation of a file named **core** in the current working directory of the process. This file contains an image of the memory of the process, together with descriptive information about the process, perhaps sufficient to reconstruct the state of the process at the receipt of the signal.

There is a potential security problem in creating a **core** file if the process was set-user-ID and the current user is not the owner of the program, if the process was set-group-ID and none of the user's groups match the group of the program, or if the user does not have permission to write in the current directory. In this situation, an implementation either should not create a **core** file or should make it unreadable by the user.

Despite the silence of this volume of IEEE Std 1003.1-200x on this feature, applications are advised not to create files named **core** because of potential conflicts in many implementations. Some historical implementations use a different name than **core** for the file, such as by appending the process ID to the filename.

Terminating a Process

It is important that the consequences of process termination as described occur regardless of whether the process called *_exit*( ) (perhaps indirectly through *exit*( )) or instead was terminated due to a signal or for some other reason. Note that in the specific case of *exit*( ) this means that the *status* argument to *exit*( ) is treated in the same way as the *status* argument to *_exit*( ).

A language other than C may have other termination primitives than the C-language *exit*( ) function, and programs written in such a language should use its native termination primitives, but those should have as part of their function the behavior of *_exit*( ) as described. Implementations in languages other than C are outside the scope of the present version of this volume of IEEE Std 1003.1-200x, however.

As required by the ISO C standard, using **return** from *main*( ) has the same behavior (other than with respect to language scope issues) as calling *exit*( ) with the returned value. Reaching the end of the *main*( ) function has the same behavior as calling *exit*(0).

A value of zero (or EXIT_SUCCESS, which is required to be zero) for the argument *status* conventionally indicates successful termination. This corresponds to the specification for *exit*( ) in the ISO C standard. The convention is followed by utilities such as *make* and various shells, which interpret a zero status from a child process as success. For this reason, applications should not call *exit*(0) or *_exit*(0) when they terminate unsuccessfully; for example, in signal-catching functions.

Historically, the implementation-defined process that inherits children whose parents have terminated without waiting on them is called *init* and has a process ID of 1.

The sending of a SIGHUP to the foreground process group when a controlling process terminates corresponds to somewhat different historical implementations. In System V, the kernel sends a SIGHUP on termination of (essentially) a controlling process. In 4.2 BSD, the kernel does not send SIGHUP in a case like this, but the termination of a controlling process is usually noticed by a system daemon, which arranges to send a SIGHUP to the foreground process group with the *vhangup*( ) function. However, in 4.2 BSD, due to the behavior of the shells that support job control, the controlling process is usually a shell with no other processes in its process group. Thus, a change to make *_exit*( ) behave this way in such systems should not cause problems with existing applications.

The termination of a process may cause a process group to become orphaned in either of two ways. The connection of a process group to its parent(s) outside of the group depends on both the parents and their children. Thus, a process group may be orphaned by the termination of the last connecting parent process outside of the group or by the termination of the last direct descendant of the parent process(es). In either case, if the termination of a process causes a process group to become orphaned, processes within the group are disconnected from their job control shell, which no longer has any information on the existence of the process group. Stopped processes within the group would languish forever. In order to avoid this problem, newly orphaned process groups that contain stopped processes are sent a SIGHUP signal and a SIGCONT signal to indicate that they have been disconnected from their session. The SIGHUP signal causes the process group members to terminate unless they are catching or ignoring SIGHUP. Under most circumstances, all of the members of the process group are stopped if any of them are stopped.

The action of sending a SIGHUP and a SIGCONT signal to members of a newly orphaned process group is similar to the action of 4.2 BSD, which sends SIGHUP and SIGCONT to each stopped child of an exiting process. If such children exit in response to the SIGHUP, any additional descendants receive similar treatment at that time. In this volume of IEEE Std 1003.1-200x, the signals are sent to the entire process group at the same time. Also, in

this volume of IEEE Std 1003.1-200x, but not in 4.2 BSD, stopped processes may be orphaned, but may be members of a process group that is not orphaned; therefore, the action taken at *_exit*( ) must consider processes other than child processes.

It is possible for a process group to be orphaned by a call to *setpgid*( ) or *setsid*( ), as well as by process termination. This volume of IEEE Std 1003.1-200x does not require sending SIGHUP and SIGCONT in those cases, because, unlike process termination, those cases are not caused accidentally by applications that are unaware of job control. An implementation can choose to send SIGHUP and SIGCONT in those cases as an extension; such an extension must be documented as required in **<signal.h>**.

The ISO / IEC 9899: 1999 standard adds the *_Exit*( ) function that results in immediate program termination without triggering signals or *atexit*( )-registered functions. In IEEE Std 1003.1-200x, this is equivalent to the *_exit*( ) function.

FUTURE DIRECTIONS

None.

SEE ALSO

atexit( ), *close*( ), *fclose*( ), *longjmp*( ), *posix_trace_shutdown*( ), *posix_trace_trid_eventid_open*( ), *semop*( ), *shmget*( ), *sigaction*( ), *wait*( ), *waitid*( ), *waitpid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Interactions with the SA_NOCLDWAIT flag and SIGCHLD signal are further clarified.

The values of *status* from *exit*( ) are better described.

Issue 6

Extensions beyond the ISO C standard are now marked.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for typed memory.

The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

- The *_Exit*( ) function is included.
- The DESCRIPTION is updated.

The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

References to the *wait3*( ) function are removed.

10190 **NAME**

10191 exp, expf, expl — exponential function

10192 **SYNOPSIS**

10193
```
#include <math.h>
```

10194
```
double exp(double x);
```
10195
```
float expf(float x);
```
10196
```
long double expl(long double x);
```

10197 **DESCRIPTION**

10198 CX The functionality described on this reference page is aligned with the ISO C standard. Any
10199 conflict between the requirements described here and the ISO C standard is unintentional. This
10200 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

10201 These functions shall compute the base-*e* exponential of *x*.

10202 An application wishing to check for error situations should set *errno* to zero and call
10203 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
10204 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
10205 zero, an error has occurred.

10206 **RETURN VALUE**

10207 Upon successful completion, these functions shall return the exponential value of *x*.

10208 If the correct value would cause overflow, a range error shall occur and *exp*( ), *expf*( ), and *expl*( )
10209 shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

10210 If the correct value would cause underflow, and is not representable, a range error may occur,
10211 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

10212 MX If *x* is NaN, a NaN shall be returned.

10213 If *x* is ±0, 1 shall be returned.

10214 If *x* is –Inf, +0 shall be returned.

10215 If *x* is +Inf, *x* shall be returned.

10216 If the correct value would cause underflow, and is representable, a range error may occur and
10217 the correct value shall be returned.

10218 **ERRORS**

10219 These functions shall fail if:

10220 Range Error The result overflows.

10221 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
10222 then *errno* shall be set to [ERANGE]. If the integer expression |
10223 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow |
10224 floating-point exception shall be raised. |

10225 These functions may fail if:

10226 Range Error The result underflows.

10227 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
10228 then *errno* shall be set to [ERANGE]. If the integer expression |
10229 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow |
10230 floating-point exception shall be raised. |

EXAMPLES

None.

APPLICATION USAGE

Note that for IEEE Std 754-1985 **double**, 709.8 < *x* implies *exp*(*x*) has overflowed. The value *x* < –708.4 implies *exp*(*x*) has underflowed.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *isnan*( ), *log*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *expf*( ) and *expl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

10256 **NAME**

10257 exp2, exp2f, exp2l — exponential base 2 functions

10258 **SYNOPSIS**

10259
```
#include <math.h>
```

10260
```
double exp2(double x);
```
10261
```
float exp2f(float x);
```
10262
```
long double exp2l(long double x);
```

10263 **DESCRIPTION**

10264 CX The functionality described on this reference page is aligned with the ISO C standard. Any
10265 conflict between the requirements described here and the ISO C standard is unintentional. This
10266 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

10267 These functions shall compute the base-2 exponential of *x*.

10268 An application wishing to check for error situations should set *errno* to zero and call
10269 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
10270 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
10271 zero, an error has occurred.

10272 **RETURN VALUE**

10273 Upon successful completion, these functions shall return 2^x.

10274 If the correct value would cause overflow, a range error shall occur and *exp2*( ), *exp2f*( ), and
10275 *exp2l*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
10276 respectively.

10277 If the correct value would cause underflow, and is not representable, a range error may occur,
10278 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

10279 MX If *x* is NaN, a NaN shall be returned.

10280 If *x* is ±0, 1 shall be returned.

10281 If *x* is –Inf, +0 shall be returned.

10282 If *x* is +Inf, *x* shall be returned.

10283 If the correct value would cause underflow, and is representable, a range error may occur and
10284 the correct value shall be returned.

10285 **ERRORS**

10286 These functions shall fail if:

10287 Range Error The result overflows.

10288 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
10289 then *errno* shall be set to [ERANGE]. If the integer expression |
10290 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow |
10291 floating-point exception shall be raised. |

10292 These functions may fail if:

10293 Range Error The result underflows.

10294 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
10295 then *errno* shall be set to [ERANGE]. If the integer expression |
10296 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow |
10297 floating-point exception shall be raised. |

10298 **EXAMPLES**

10299 None.

10300 **APPLICATION USAGE**

10301 For IEEE Std 754-1985 **double**, 1024 <= *x* implies *exp2*(*x*) has overflowed. The value *x* < –1022
10302 implies *exp*(*x*) has underflowed.

10303 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
10304 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

10305 **RATIONALE**

10306 None.

10307 **FUTURE DIRECTIONS**

10308 None.

10309 **SEE ALSO**

10310 *exp*( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *log*( ), the Base Definitions volume of |
10311 IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, |
10312 **<math.h>**

10313 **CHANGE HISTORY**

10314 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

10315 **NAME**

10316 expm1, expm1f, expm1l — compute exponential functions

10317 **SYNOPSIS**

10318
```
#include <math.h>
```

10319
```
double expm1(double x);
```
10320
```
float expm1f(float x);
```
10321
```
long double expm1l(long double x);
```

10322 **DESCRIPTION**

10323 CX The functionality described on this reference page is aligned with the ISO C standard. Any
10324 conflict between the requirements described here and the ISO C standard is unintentional. This
10325 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

10326 These functions shall compute e^x–1.0.

10327 An application wishing to check for error situations should set *errno* to zero and call
10328 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
10329 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
10330 zero, an error has occurred.

10331 **RETURN VALUE**

10332 Upon successful completion, these functions return e^x–1.0.

10333 If the correct value would cause overflow, a range error shall occur and *expm1*( ), *expm1f*( ), and
10334 *expm1l*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
10335 respectively.

10336 MX If *x* is NaN, a NaN shall be returned.

10337 If *x* is ±0, ±0 shall be returned.

10338 If *x* is –Inf, –1 shall be returned.

10339 If *x* is +Inf, *x* shall be returned.

10340 If *x* is subnormal, a range error may occur and *x* should be returned.

10341 **ERRORS**

10342 These functions shall fail if:

10343 Range Error The result overflows.

10344 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
10345 then *errno* shall be set to [ERANGE]. If the integer expression |
10346 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow |
10347 floating-point exception shall be raised. |

10348 These functions may fail if:

10349 MX Range Error The value of *x* is subnormal.

10350 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
10351 then *errno* shall be set to [ERANGE]. If the integer expression |
10352 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow |
10353 floating-point exception shall be raised. |

EXAMPLES

None.

APPLICATION USAGE

The value of *expm1*(*x*) may be more accurate than *exp*(*x*)–1.0 for small values of *x*.

The *expm1*( ) and *log1p*( ) functions are useful for financial calculations of $((1+x)^n-1)/x$, namely:

```
expm1(n * log1p(x))/x
```

when *x* is very small (for example, when calculating small daily interest rates). These functions also simplify writing accurate inverse hyperbolic functions.

For IEEE Std 754-1985 **double**, 709.8 < *x* implies *expm1*(*x*) has overflowed.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exp( ), *feclearexcept*( ), *fetestexcept*( ), *ilogb*( ), *log1p*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *expm1f*( ) and *expm1l*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The *expm1*( ) function is no longer marked as an extension.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

10385 **NAME**

10386 fabs, fabsf, fabsl — absolute value function

10387 **SYNOPSIS**

10388 `#include <math.h>`

10389 `double fabs(double x);`
10390 `float fabsf(float x);`
10391 `long double fabsl(long double x);`

10392 **DESCRIPTION**

10393 CX The functionality described on this reference page is aligned with the ISO C standard. Any
10394 conflict between the requirements described here and the ISO C standard is unintentional. This
10395 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

10396 These functions shall compute the absolute value of their argument x, $|x|$.

10397 **RETURN VALUE**

10398 Upon successful completion, these functions shall return the absolute value of x.

10399 MX If x is NaN, a NaN shall be returned.

10400 If x is ±0, +0 shall be returned.

10401 If x is ±Inf, +Inf shall be returned.

10402 **ERRORS**

10403 No errors are defined.

10404 **EXAMPLES**

10405 None.

10406 **APPLICATION USAGE**

10407 None.

10408 **RATIONALE**

10409 None.

10410 **FUTURE DIRECTIONS**

10411 None.

10412 **SEE ALSO**

10413 *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

10414 **CHANGE HISTORY**

10415 First released in Issue 1. Derived from Issue 1 of the SVID.

10416 **Issue 5**

10417 The DESCRIPTION is updated to indicate how an application should check for an error. This
10418 text was previously published in the APPLICATION USAGE section.

10419 **Issue 6**

10420 The *fabsf*( ) and *fabsl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

10421 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
10422 revised to align with the ISO / IEC 9899: 1999 standard.

10423 IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are
10424 marked.

10425 **NAME**

10426 fattach — attach a STREAMS-based file descriptor to a file in the file system name space
10427 (**STREAMS**)

10428 **SYNOPSIS**

10429 XSR
```
#include <stropts.h>

int fattach(int fildes, const char *path);
```
10430
10431

10432 **DESCRIPTION**

10433 The *fattach*( ) function shall attach a STREAMS-based file descriptor to a file, effectively
10434 associating a pathname with *fildes*. The application shall ensure that the *fildes* argument is a
10435 valid open file descriptor associated with a STREAMS file. The *path* argument points to a
10436 pathname of an existing file. The application shall have the appropriate privileges, or is the
10437 owner of the file named by *path* and has write permission. A successful call to *fattach*( ) shall
10438 cause all pathnames that name the file named by *path* to name the STREAMS file associated with
10439 *fildes*, until the STREAMS file is detached from the file. A STREAMS file can be attached to more
10440 than one file and can have several pathnames associated with it.

10441 The attributes of the named STREAMS file shall be initialized as follows: the permissions, user
10442 ID, group ID, and times are set to those of the file named by *path*, the number of links is set to 1,
10443 and the size and device identifier are set to those of the STREAMS file associated with *fildes*. If
10444 any attributes of the named STREAMS file are subsequently changed (for example, by *chmod*( )),
10445 neither the attributes of the underlying file nor the attributes of the STREAMS file to which *fildes*
10446 refers shall be affected.

10447 File descriptors referring to the underlying file, opened prior to an *fattach*( ) call, shall continue to
10448 refer to the underlying file.

10449 **RETURN VALUE**

10450 Upon successful completion, *fattach*( ) shall return 0. Otherwise, –1 shall be returned and *errno*
10451 set to indicate the error.

10452 **ERRORS**

10453 The *fattach*( ) function shall fail if:

10454 [EACCES] Search permission is denied for a component of the path prefix, or the process
10455 is the owner of *path* but does not have write permissions on the file named by
10456 *path*.

10457 [EBADF] The *fildes* argument is not a valid open file descriptor.

10458 [EBUSY] The file named by *path* is currently a mount point or has a STREAMS file
10459 attached to it.

10460 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
10461 argument.

10462 [ENAMETOOLONG]
10463 The size of *path* exceeds {PATH_MAX} or a component of *path* is longer than
10464 {NAME_MAX}.

10465 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

10466 [ENOTDIR] A component of the path prefix is not a directory.

10467 [EPERM] The effective user ID of the process is not the owner of the file named by *path*
10468 and the process does not have appropriate privilege.

The *fattach*( ) function may fail if:

[EINVAL] The *fildes* argument does not refer to a STREAMS file.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

[EXDEV] A link to a file on another file system was attempted.

EXAMPLES

Attaching a File Descriptor to a File

In the following example, *fd* refers to an open STREAMS file. The call to *fattach*( ) associates this STREAM with the file **/tmp/named-STREAM**, such that any future calls to open **/tmp/named-STREAM**, prior to breaking the attachment via a call to *fdetach*( ), will instead create a new file handle referring to the STREAMS file associated with *fd*.

```
#include <stropts.h>
...
    int fd;
    char *filename = "/tmp/named-STREAM";
    int ret;

    ret = fattach(fd, filename);
```

APPLICATION USAGE

The *fattach*( ) function behaves similarly to the traditional *mount*( ) function in the way a file is temporarily replaced by the root directory of the mounted file system. In the case of *fattach*( ), the replaced file need not be a directory and the replacing file is a STREAMS file.

RATIONALE

The file attributes of a file which has been the subject of an *fattach*( ) call are specifically set because of an artefact of the original implementation. The internal mechanism was the same as for the *mount*( ) function. Since *mount*( ) is typically only applied to directories, the effects when applied to a regular file are a little surprising, especially as regards the link count which rigidly remains one, even if there were several links originally and despite the fact that all original links refer to the STREAM as long as the *fattach*( ) remains in effect.

FUTURE DIRECTIONS

None.

SEE ALSO

fdetach( ), *isastream*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The [EXDEV] error is added to the list of optional errors in the ERRORS section.

Issue 6

This function is marked as part of the XSI STREAMS Option Group.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

10514 **NAME**

10515 fchdir — change working directory

10516 **SYNOPSIS**

10517 XSI
```
#include <unistd.h>
```
10518
```
int fchdir(int fildes);
```
10519

10520 **DESCRIPTION**

10521 The *fchdir*( ) function shall be equivalent to *chdir*( ) except that the directory that is to be the new |
10522 current working directory is specified by the file descriptor *fildes*.

10523 A conforming application can obtain a file descriptor for a file of type directory using *open*( ),
10524 provided that the file status flags and access modes do not contain O_WRONLY or O_RDWR.

10525 **RETURN VALUE**

10526 Upon successful completion, *fchdir*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to
10527 indicate the error. On failure the current working directory shall remain unchanged.

10528 **ERRORS**

10529 The *fchdir*( ) function shall fail if:

10530 [EACCES] Search permission is denied for the directory referenced by *fildes*.

10531 [EBADF] The *fildes* argument is not an open file descriptor.

10532 [ENOTDIR] The open file descriptor *fildes* does not refer to a directory.

10533 The *fchdir*( ) may fail if:

10534 [EINTR] A signal was caught during the execution of *fchdir*( ).

10535 [EIO] An I/O error occurred while reading from or writing to the file system.

10536 **EXAMPLES**

10537 None.

10538 **APPLICATION USAGE**

10539 None.

10540 **RATIONALE**

10541 None.

10542 **FUTURE DIRECTIONS**

10543 None.

10544 **SEE ALSO**

10545 *chdir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

10546 **CHANGE HISTORY**

10547 First released in Issue 4, Version 2.

10548 **Issue 5**

10549 Moved from X/OPEN UNIX extension to BASE.

10550 **NAME**

10551 fchmod — change mode of a file

10552 **SYNOPSIS**

10553 `#include <sys/stat.h>`

10554 `int fchmod(int fildes, mode_t mode);`

10555 **DESCRIPTION**

10556 The *fchmod*( ) function shall be equivalent to *chmod*( ) except that the file whose permissions are |
10557 changed is specified by the file descriptor *fildes*.

10558 SHM If *fildes* references a shared memory object, the *fchmod*( ) function need only affect the S_IRUSR,
10559 S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits.

10560 TYM If *fildes* references a typed memory object, the behavior of *fchmod*( ) is unspecified. |

10561 If *fildes* refers to a socket, the behavior of *fchmod*( ) is unspecified. |

10562 XSR If *fildes* refers to a STREAM (which is *fattach*( )-ed into the file system name space) the call |
10563 returns successfully, doing nothing. |

10564 **RETURN VALUE**

10565 Upon successful completion, *fchmod*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to
10566 indicate the error.

10567 **ERRORS**

10568 The *fchmod*( ) function shall fail if:

10569 [EBADF] The *fildes* argument is not an open file descriptor.

10570 [EPERM] The effective user ID does not match the owner of the file and the process
10571 does not have appropriate privilege.

10572 [EROFS] The file referred to by *fildes* resides on a read-only file system.

10573 The *fchmod*( ) function may fail if:

10574 XSI [EINTR] The *fchmod*( ) function was interrupted by a signal.

10575 XSI [EINVAL] The value of the *mode* argument is invalid.

10576 [EINVAL] The *fildes* argument refers to a pipe and the implementation disallows
10577 execution of *fchmod*( ) on a pipe.

10578 **EXAMPLES**

10579 **Changing the Current Permissions for a File**

10580 The following example shows how to change the permissions for a file named **/home/cnd/mod1**
10581 so that the owner and group have read/write/execute permissions, but the world only has
10582 read/write permissions.

```
10583 #include <sys/stat.h>
10584 #include <fcntl.h>

10585 mode_t mode;
10586 int    fildes;
10587 ...
10588 fildes = open("/home/cnd/mod1", O_RDWR);
10589 fchmod(fildes, S_IRWXU | S_IRWXG | S_IROTH | S_IWOTH);
```

10590 **APPLICATION USAGE**

10591 None.

10592 **RATIONALE**

10593 None.

10594 **FUTURE DIRECTIONS**

10595 None.

10596 **SEE ALSO**

10597 *chmod*( ), *chown*( ), *creat*( ), *fcntl*( ), *fstatvfs*( ), *mknod*( ), *open*( ), *read*( ), *stat*( ), *write*( ), the Base
10598 Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**

10599 **CHANGE HISTORY**

10600 First released in Issue 4, Version 2.

10601 **Issue 5**

10602 Moved from X/OPEN UNIX extension to BASE and aligned with *fchmod*( ) in the POSIX
10603 Realtime Extension. Specifically, the second paragraph of the DESCRIPTION is added and a
10604 second instance of [EINVAL] is defined in the list of optional errors.

10605 **Issue 6**

10606 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by stating that *fchmod*( )
10607 behavior is unspecified for typed memory objects.

10608 **NAME**

10609 fchown — change owner and group of a file

10610 **SYNOPSIS**

10611 `#include <unistd.h>`

10612 `int fchown(int fildes, uid_t owner, gid_t group);`

10613 **DESCRIPTION**

10614 The *fchown*( ) function shall be equivalent to *chown*( ) except that the file whose owner and group |
10615 are changed is specified by the file descriptor *fildes*.

10616 **RETURN VALUE**

10617 Upon successful completion, *fchown*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to
10618 indicate the error.

10619 **ERRORS**

10620 The *fchown*( ) function shall fail if:

10621 [EBADF] The *fildes* argument is not an open file descriptor.

10622 [EPERM] The effective user ID does not match the owner of the file or the process does
10623 not have appropriate privilege and _POSIX_CHOWN_RESTRICTED indicates
10624 that such privilege is required.

10625 [EROFS] The file referred to by *fildes* resides on a read-only file system.

10626 The *fchown*( ) function may fail if:

10627 [EINVAL] The owner or group ID is not a value supported by the implementation. The
10628 XSR *fildes* argument refers to a pipe or socket or an *fattach*( )-ed STREAM and the |
10629 implementation disallows execution of *fchown*( ) on a pipe. |

10630 [EIO] A physical I/O error has occurred.

10631 [EINTR] The *fchown*( ) function was interrupted by a signal which was caught.

10632 **EXAMPLES**

10633 **Changing the Current Owner of a File**

10634 The following example shows how to change the owner of a file named **/home/cnd/mod1** to
10635 ‘‘jones’’ and the group to ‘‘cnd’’.

10636 The numeric value for the user ID is obtained by extracting the user ID from the user database
10637 entry associated with ‘‘jones’’. Similarly, the numeric value for the group ID is obtained by
10638 extracting the group ID from the group database entry associated with ‘‘cnd’’. This example
10639 assumes the calling program has appropriate privileges.

```
10640 #include <sys/types.h>
10641 #include <unistd.h>
10642 #include <fcntl.h>
10643 #include <pwd.h>
10644 #include <grp.h>

10645 struct passwd *pwd;
10646 struct group  *grp;
10647 int            fildes;
10648 ...
10649 fildes = open("/home/cnd/mod1", O_RDWR);
10650 pwd = getpwnam("jones");
```

```
10651       grp = getgrnam("cnd");
10652       fchown(fildes, pwd->pw_uid, grp->gr_gid);
```

10653 **APPLICATION USAGE**

10654 None.

10655 **RATIONALE**

10656 None.

10657 **FUTURE DIRECTIONS**

10658 None.

10659 **SEE ALSO**

10660 *chown*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

10661 **CHANGE HISTORY**

10662 First released in Issue 4, Version 2.

10663 **Issue 5**

10664 Moved from X/OPEN UNIX extension to BASE.

10665 **Issue 6**

10666 The following changes were made to align with the IEEE P1003.1a draft standard:

10667 • Clarification is added that a call to *fchown*( ) may not be allowed on a pipe.

10668 The *fchown*( ) function is now defined as mandatory.

10669 **NAME**

10670 fclose — close a stream

10671 **SYNOPSIS**

10672 `#include <stdio.h>`

10673 `int fclose(FILE *stream);`

10674 **DESCRIPTION**

10675 CX The functionality described on this reference page is aligned with the ISO C standard. Any
10676 conflict between the requirements described here and the ISO C standard is unintentional. This
10677 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

10678 The *fclose*( ) function shall cause the stream pointed to by *stream* to be flushed and the associated
10679 file to be closed. Any unwritten buffered data for the stream shall be written to the file; any
10680 unread buffered data shall be discarded. Whether or not the call succeeds, the stream shall be
10681 disassociated from the file and any buffer set by the *setbuf*( ) or *setvbuf*( ) function shall be
10682 disassociated from the stream. If the associated buffer was automatically allocated, it shall be
10683 deallocated. |

10684 CX The *fclose*( ) function shall mark for update the *st_ctime* and *st_mtime* fields of the underlying file, |
10685 if the stream was writable, and if buffered data remains that has not yet been written to the file.
10686 The *fclose*( ) function shall perform the equivalent of a *close*( ) on the file descriptor that is
10687 associated with the stream pointed to by *stream*.

10688 After the call to *fclose*( ), any use of *stream* results in undefined behavior.

10689 **RETURN VALUE**

10690 CX Upon successful completion, *fclose*( ) shall return 0; otherwise, it shall return EOF and set *errno* to
10691 indicate the error.

10692 **ERRORS**

10693 The *fclose*( ) function shall fail if:

10694 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
10695 process would be delayed in the write operation.

10696 CX [EBADF] The file descriptor underlying stream is not valid.

10697 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

10698 XSI [EFBIG] An attempt was made to write a file that exceeds the process' file size limit.

10699 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
10700 offset maximum associated with the corresponding stream.

10701 CX [EINTR] The *fclose*( ) function was interrupted by a signal.

10702 CX [EIO] The process is a member of a background process group attempting to write
10703 to its controlling terminal, TOSTOP is set, the process is neither ignoring nor
10704 blocking SIGTTOU, and the process group of the process is orphaned. This
10705 error may also be returned under implementation-defined conditions.

10706 CX [ENOSPC] There was no free space remaining on the device containing the file.

10707 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
10708 any process. A SIGPIPE signal shall also be sent to the thread.

10709 The *fclose*( ) function may fail if:

10710 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
10711 capabilities of the device.

10712 **EXAMPLES**

10713 None.

10714 **APPLICATION USAGE**

10715 None.

10716 **RATIONALE**

10717 None.

10718 **FUTURE DIRECTIONS**

10719 None.

10720 **SEE ALSO**

10721 *close*( ), *fopen*( ), *getrlimit*( ), *ulimit*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
10722 **<stdio.h>**

10723 **CHANGE HISTORY**

10724 First released in Issue 1. Derived from Issue 1 of the SVID.

10725 **Issue 5**

10726 Large File Summit extensions are added.

10727 **Issue 6**

10728 Extensions beyond the ISO C standard are now marked.

10729 The following new requirements on POSIX implementations derive from alignment with the
10730 Single UNIX Specification:

10731 - The [EFBIG] error is added as part of the large file support extensions.

10732 - The [ENXIO] optional error condition is added.

10733 The DESCRIPTION is updated to note that the stream and any buffer are disassociated whether
10734 or not the call succeeds. This is for alignment with the ISO / IEC 9899: 1999 standard.

10735 **NAME**

10736 fcntl — file control

10737 **SYNOPSIS**

10738 OH
```
#include <unistd.h>
```
10739
```
#include <fcntl.h>
```
10740
```
int fcntl(int fildes, int cmd, ...);
```

10741 **DESCRIPTION**

10742 The *fcntl*( ) function shall perform the operations described below on open files. The *fildes* |
10743 argument is a file descriptor.

10744 The available values for *cmd* are defined in **<fcntl.h>** and are as follows: |

10745 F_DUPFD Return a new file descriptor which shall be the lowest numbered available |
10746 (that is, not already open) file descriptor greater than or equal to the third |
10747 argument, *arg*, taken as an integer of type **int**. The new file descriptor shall |
10748 refer to the same open file description as the original file descriptor, and shall |
10749 share any locks. The FD_CLOEXEC flag associated with the new file |
10750 descriptor shall be cleared to keep the file open across calls to one of the *exec* |
10751 functions.

10752 F_GETFD Get the file descriptor flags defined in **<fcntl.h>** that are associated with the
10753 file descriptor *fildes*. File descriptor flags are associated with a single file
10754 descriptor and do not affect other file descriptors that refer to the same file.

10755 F_SETFD Set the file descriptor flags defined in **<fcntl.h>**, that are associated with *fildes*,
10756 to the third argument, *arg*, taken as type **int**. If the FD_CLOEXEC flag in the
10757 third argument is 0, the file shall remain open across the *exec* functions;
10758 otherwise, the file shall be closed upon successful execution of one of the *exec*
10759 functions.

10760 F_GETFL Get the file status flags and file access modes, defined in **<fcntl.h>**, for the file
10761 description associated with *fildes*. The file access modes can be extracted from
10762 the return value using the mask O_ACCMODE, which is defined in **<fcntl.h>**. |
10763 File status flags and file access modes are associated with the file description
10764 and do not affect other file descriptors that refer to the same file with different
10765 open file descriptions.

10766 F_SETFL Set the file status flags, defined in **<fcntl.h>**, for the file description associated
10767 with *fildes* from the corresponding bits in the third argument, *arg*, taken as
10768 type **int**. Bits corresponding to the file access mode and the file creation flags, |
10769 as defined in **<fcntl.h>**, that are set in *arg* shall be ignored. If any bits in *arg* |
10770 other than those mentioned here are changed by the application, the result is
10771 unspecified.

10772 F_GETOWN If *fildes* refers to a socket, get the process or process group ID specified to
10773 receive SIGURG signals when out-of-band data is available. Positive values
10774 indicate a process ID; negative values, other than –1, indicate a process group
10775 ID. If *fildes* does not refer to a socket, the results are unspecified.

10776 F_SETOWN If *fildes* refers to a socket, set the process or process group ID specified to
10777 receive SIGURG signals when out-of-band data is available, using the value of
10778 the third argument, *arg*, taken as type **int**. Positive values indicate a process
10779 ID; negative values, other than –1, indicate a process group ID. If *fildes* does
10780 not refer to a socket, the results are unspecified.

The following values for *cmd* are available for advisory record locking. Record locking shall be supported for regular files, and may be supported for other files.

F_GETLK — Get the first lock which blocks the lock description pointed to by the third argument, *arg*, taken as a pointer to type **struct flock**, defined in **<fcntl.h>**. The information retrieved shall overwrite the information passed to *fcntl*( ) in the structure **flock**. If no lock is found that would prevent this lock from being created, then the structure shall be left unchanged except for the lock type which shall be set to F_UNLCK.

F_SETLK — Set or clear a file segment lock according to the lock description pointed to by the third argument, *arg*, taken as a pointer to type **struct flock**, defined in **<fcntl.h>**. F_SETLK can establish shared (or read) locks (F_RDLCK) or exclusive (or write) locks (F_WRLCK), as well as to remove either type of lock (F_UNLCK). F_RDLCK, F_WRLCK, and F_UNLCK are defined in **<fcntl.h>**. If a shared or exclusive lock cannot be set, *fcntl*( ) shall return immediately with a return value of −1.

F_SETLKW — This command shall be equivalent to F_SETLK except that if a shared or exclusive lock is blocked by other locks, the thread shall wait until the request can be satisfied. If a signal that is to be caught is received while *fcntl*( ) is waiting for a region, *fcntl*( ) shall be interrupted. Upon return from the signal handler, *fcntl*( ) shall return −1 with *errno* set to [EINTR], and the lock operation shall not be done.

Additional implementation-defined values for *cmd* may be defined in **<fcntl.h>**. Their names shall start with F_.

When a shared lock is set on a segment of a file, other processes shall be able to set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock shall fail if the file descriptor was not opened with read access.

An exclusive lock shall prevent any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock shall fail if the file descriptor was not opened with write access.

The structure **flock** describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), and process ID (*l_pid*) of the segment of the file to be affected.

The value of *l_whence* is SEEK_SET, SEEK_CUR, or SEEK_END, to indicate that the relative offset *l_start* bytes shall be measured from the start of the file, current position, or end of the file, respectively. The value of *l_len* is the number of consecutive bytes to be locked. The value of *l_len* may be negative (where the definition of **off_t** permits negative values of *l_len*). The *l_pid* field is only used with F_GETLK to return the process ID of the process holding a blocking lock. After a successful F_GETLK request, when a blocking lock is found, the values returned in the **flock** structure shall be as follows:

l_type — Type of blocking lock found.

l_whence — SEEK_SET.

l_start — Start of the blocking lock.

l_len — Length of the blocking lock.

l_pid — Process ID of the process that holds the blocking lock.

If the command is F_SETLKW and the process must wait for another process to release a lock, then the range of bytes to be locked shall be determined before the *fcntl*( ) function blocks. If the file size or file descriptor seek offset change while *fcntl*( ) is blocked, this shall not affect the range of bytes locked.

If *l_len* is positive, the area affected shall start at *l_start* and end at *l_start*+*l_len*−1. If *l_len* is negative, the area affected shall start at *l_start*+*l_len* and end at *l_start*−1. Locks may start and extend beyond the current end of a file, but shall not extend before the beginning of the file. A lock shall be set to extend to the largest possible value of the file offset for that file by setting *l_len* to 0. If such a lock also has *l_start* set to 0 and *l_whence* is set to SEEK_SET, the whole file shall be locked.

There shall be at most one type of lock set for each byte in the file. Before a successful return from an F_SETLK or an F_SETLKW request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region shall be replaced by the new lock type. As specified above under the descriptions of shared locks and exclusive locks, an F_SETLK or an F_SETLKW request (respectively) shall fail or block when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

All locks associated with a file for a given process shall be removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock another process' locked region. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, *fcntl*( ) shall fail with an [EDEADLK] error.

An unlock (F_UNLCK) request in which *l_len* is non-zero and the offset of the last byte of the requested segment is the maximum value for an object of type **off_t**, when the process has an existing lock in which *l_len* is 0 and which includes the last byte of the requested segment, shall be treated as a request to unlock from the start of the requested segment with an *l_len* equal to 0. Otherwise, an unlock (F_UNLCK) request shall attempt to unlock only the requested segment.

SHM When the file descriptor *fildes* refers to a shared memory object, the behavior of *fcntl*( ) shall be the same as for a regular file except the effect of the following values for the argument *cmd* shall be unspecified: F_SETFL, F_GETLK, F_SETLK, and F_SETLKW.

TYM If *fildes* refers to a typed memory object, the result of the *fcntl*( ) function is unspecified.

RETURN VALUE

Upon successful completion, the value returned shall depend on *cmd* as follows:

F_DUPFD	A new file descriptor.
F_GETFD	Value of flags defined in **<fcntl.h>**. The return value shall not be negative.
F_SETFD	Value other than −1.
F_GETFL	Value of file status flags and access modes. The return value is not negative.
F_SETFL	Value other than −1.
F_GETLK	Value other than −1.
F_SETLK	Value other than −1.
F_SETLKW	Value other than −1.
F_GETOWN	Value of the socket owner process or process group; this will not be −1.

F_SETOWN Value other than –1.

Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *fcntl*( ) function shall fail if:

[EACCES] or [EAGAIN]
The *cmd* argument is F_SETLK; the type of lock (*l_type*) is a shared (F_RDLCK) or exclusive (F_WRLCK) lock and the segment of a file to be locked is already exclusive-locked by another process, or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

[EBADF] The *fildes* argument is not a valid open file descriptor, or the argument *cmd* is F_SETLK or F_SETLKW, the type of lock, *l_type*, is a shared lock (F_RDLCK), and *fildes* is not a valid file descriptor open for reading, or the type of lock *l_type*, is an exclusive lock (F_WRLCK), and *fildes* is not a valid file descriptor open for writing.

[EINTR] The *cmd* argument is F_SETLKW and the function was interrupted by a signal.

[EINVAL] The *cmd* argument is invalid, or the *cmd* argument is F_DUPFD and *arg* is negative or greater than or equal to {OPEN_MAX}, or the *cmd* argument is F_GETLK, F_SETLK, or F_SETLKW and the data pointed to by *arg* is not valid, or *fildes* refers to a file that does not support locking.

[EMFILE] The argument *cmd* is F_DUPFD and {OPEN_MAX} file descriptors are currently open in the calling process, or no file descriptors greater than or equal to *arg* are available.

[ENOLCK] The argument *cmd* is F_SETLK or F_SETLKW and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.

[EOVERFLOW] One of the values to be returned cannot be represented correctly.

[EOVERFLOW] The *cmd* argument is F_GETLK, F_SETLK, or F_SETLKW and the smallest or, if *l_len* is non-zero, the largest offset of any byte in the requested segment cannot be represented correctly in an object of type **off_t**.

The *fcntl*( ) function may fail if:

[EDEADLK] The *cmd* argument is F_SETLKW, the lock is blocked by some lock from another process and putting the calling process to sleep, waiting for that lock to become free would cause a deadlock.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The ellipsis in the SYNOPSIS is the syntax specified by the ISO C standard for a variable number of arguments. It is used because System V uses pointers for the implementation of file locking functions.

The *arg* values to F_GETFD, F_SETFD, F_GETFL, and F_SETFL all represent flag values to allow for future growth. Applications using these functions should do a read-modify-write operation

on them, rather than assuming that only the values defined by this volume of IEEE Std 1003.1-200x are valid. It is a common error to forget this, particularly in the case of F_SETFD.

This volume of IEEE Std 1003.1-200x permits concurrent read and write access to file data using the *fcntl*( ) function; this is a change from the 1984 /usr/group standard and early proposals. Without concurrency controls, this feature may not be fully utilized without occasional loss of data.

Data losses occur in several ways. One case occurs when several processes try to update the same record, without sequencing controls; several updates may occur in parallel and the last writer ''wins''. Another case is a bit-tree or other internal list-based database that is undergoing reorganization. Without exclusive use to the tree segment by the updating process, other reading processes chance getting lost in the database when the index blocks are split, condensed, inserted, or deleted. While *fcntl*( ) is useful for many applications, it is not intended to be overly general and does not handle the bit-tree example well.

This facility is only required for regular files because it is not appropriate for many devices such as terminals and network connections.

Since *fcntl*( ) works with ''any file descriptor associated with that file, however it is obtained'', the file descriptor may have been inherited through a *fork*( ) or *exec* operation and thus may affect a file that another process also has open.

The use of the open file description to identify what to lock requires extra calls and presents problems if several processes are sharing an open file description, but there are too many implementations of the existing mechanism for this volume of IEEE Std 1003.1-200x to use different specifications.

Another consequence of this model is that closing any file descriptor for a given file (whether or not it is the same open file description that created the lock) causes the locks on that file to be relinquished for that process. Equivalently, any close for any file/process pair relinquishes the locks owned on that file for that process. But note that while an open file description may be shared through *fork*( ), locks are not inherited through *fork*( ). Yet locks may be inherited through one of the *exec* functions.

The identification of a machine in a network environment is outside of the scope of this volume of IEEE Std 1003.1-200x. Thus, an *l_sysid* member, such as found in System V, is not included in the locking structure.

Changing of lock types can result in a previously locked region being split into smaller regions. |

Mandatory locking was a major feature of the 1984 /usr/group standard.

For advisory file record locking to be effective, all processes that have access to a file must cooperate and use the advisory mechanism before doing I/O on the file. Enforcement-mode record locking is important when it cannot be assumed that all processes are cooperating. For example, if one user uses an editor to update a file at the same time that a second user executes another process that updates the same file and if only one of the two processes is using advisory locking, the processes are not cooperating. Enforcement-mode record locking would protect against accidental collisions.

Secondly, advisory record locking requires a process using locking to bracket each I/O operation with lock (or test) and unlock operations. With enforcement-mode file and record locking, a process can lock the file once and unlock when all I/O operations have been completed. Enforcement-mode record locking provides a base that can be enhanced; for example, with sharable locks. That is, the mechanism could be enhanced to allow a process to lock a file so other processes could read it, but none of them could write it.

Mandatory locks were omitted for several reasons:

1. Mandatory lock setting was done by multiplexing the set-group-ID bit in most implementations; this was confusing, at best.
2. The relationship to file truncation as supported in 4.2 BSD was not well specified.
3. Any publicly readable file could be locked by anyone. Many historical implementations keep the password database in a publicly readable file. A malicious user could thus prohibit logins. Another possibility would be to hold open a long-distance telephone line.
4. Some demand-paged historical implementations offer memory mapped files, and enforcement cannot be done on that type of file.

Since sleeping on a region is interrupted with any signal, *alarm*( ) may be used to provide a timeout facility in applications requiring it. This is useful in deadlock detection. Since implementation of full deadlock detection is not always feasible, the [EDEADLK] error was made optional.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *exec*, *open*( ), *sigaction*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<signal.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the DESCRIPTION, sentences describing behavior when *l_len* is negative are now mandated, and the description of unlock (F_UNLOCK) when *l_len* is non-negative is mandated.
- In the ERRORS section, the [EINVAL] error condition has the case mandated when the *cmd* is invalid, and two [EOVERFLOW] error conditions are added.

The F_GETOWN and F_SETOWN values are added for sockets.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Clarification is added that the extent of the bytes locked is determined prior to the blocking action.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that *fcntl*( ) results are unspecified for typed memory objects.

11001 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

11002 **NAME**

11003 fcvt — convert a floating-point number to a string (**LEGACY**)

11004 **SYNOPSIS**

11005 XSI
```
#include <stdlib.h>

char *fcvt(double value, int ndigit, int *restrict decpt,
    int *restrict sign);
```
11006
11007
11008

11009 **DESCRIPTION**

11010 Refer to *ecvt*( ).

11011 **NAME**

11012 fdatasync — synchronize the data of a file (**REALTIME**)

11013 **SYNOPSIS**

11014 SIO
```
#include <unistd.h>
```

11015
```
int fdatasync(int fildes);
```

11016

11017 **DESCRIPTION**

11018 The *fdatasync*( ) function shall force all currently queued I/O operations associated with the file
11019 indicated by file descriptor *fildes* to the synchronized I/O completion state.

11020 The functionality shall be equivalent to *fsync*( ) with the symbol _POSIX_SYNCHRONIZED_IO |
11021 defined, with the exception that all I/O operations shall be completed as defined for |
11022 synchronized I/O data integrity completion.

11023 **RETURN VALUE**

11024 If successful, the *fdatasync*( ) function shall return the value 0; otherwise, the function shall return
11025 the value –1 and set *errno* to indicate the error. If the *fdatasync*( ) function fails, outstanding I/O
11026 operations are not guaranteed to have been completed.

11027 **ERRORS**

11028 The *fdatasync*( ) function shall fail if:

11029 [EBADF] The *fildes* argument is not a valid file descriptor open for writing.

11030 [EINVAL] This implementation does not support synchronized I/O for this file.

11031 In the event that any of the queued I/O operations fail, *fdatasync*( ) shall return the error
11032 conditions defined for *read*( ) and *write*( ).

11033 **EXAMPLES**

11034 None.

11035 **APPLICATION USAGE**

11036 None.

11037 **RATIONALE**

11038 None.

11039 **FUTURE DIRECTIONS**

11040 None.

11041 **SEE ALSO**

11042 *aio_fsync*( ), *fcntl*( ), *fsync*( ), *open*( ), *read*( ), *write*( ), the Base Definitions volume of
11043 IEEE Std 1003.1-200x, **<unistd.h>**

11044 **CHANGE HISTORY**

11045 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

11046 **Issue 6**

11047 The [ENOSYS] error condition has been removed as stubs need not be provided if an
11048 implementation does not support the Synchronized Input and Output option.

11049 The *fdatasync*( ) function is marked as part of the Synchronized Input and Output option.

NAME

fdetach — detach a name from a STREAMS-based file descriptor (**STREAMS**)

SYNOPSIS

XSR
```
#include <stropts.h>

int fdetach(const char *path);
```

DESCRIPTION

The *fdetach*( ) function shall detach a STREAMS-based file from the file to which it was attached by a previous call to *fattach*( ). The *path* argument points to the pathname of the attached STREAMS file. The process shall have appropriate privileges or be the owner of the file. A successful call to *fdetach*( ) shall cause all pathnames that named the attached STREAMS file to again name the file to which the STREAMS file was attached. All subsequent operations on *path* shall operate on the underlying file and not on the STREAMS file.

All open file descriptions established while the STREAMS file was attached to the file referenced by *path* shall still refer to the STREAMS file after the *fdetach*( ) has taken effect.

If there are no open file descriptors or other references to the STREAMS file, then a successful call to *fdetach*( ) shall have be equivalent to performing the last *close*( ) on the attached file.

RETURN VALUE

Upon successful completion, *fdetach*( ) shall return 0; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *fdetach*( ) function shall fail if:

[EACCES] Search permission is denied on a component of the path prefix.

[EINVAL] The *path* argument names a file that is not currently attached.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[ENAMETOOLONG]
The size of a pathname exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The effective user ID is not the owner of *path* and the process does not have appropriate privileges.

The *fdetach*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

EXAMPLES

Detaching a File

The following example detaches the STREAMS-based file **/tmp/named-STREAM** from the file to which it was attached by a previous, successful call to *fattach*( ). Subsequent calls to open this file refer to the underlying file, not to the STREAMS file.

```
#include <stropts.h>
...
    char *filename = "/tmp/named-STREAM";
    int ret;

    ret = fdetach(filename);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fattach( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

11115 **NAME**

11116 fdim, fdimf, fdiml — compute positive difference between two floating-point numbers

11117 **SYNOPSIS**

11118
```
#include <math.h>
```

11119
```
double fdim(double x, double y);
```
11120
```
float fdimf(float x, float y);
```
11121
```
long double fdiml(long double x, long double y);
```

11122 **DESCRIPTION**

11123 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11124 conflict between the requirements described here and the ISO C standard is unintentional. This
11125 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11126 These functions shall determine the positive difference between their arguments. If *x* is greater
11127 than *y*, *x*–*y* is returned. If *x* is less than or equal to *y*, +0 is returned.

11128 An application wishing to check for error situations should set *errno* to zero and call
11129 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
11130 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
11131 zero, an error has occurred.

11132 **RETURN VALUE**

11133 Upon successful completion, these functions shall return the positive difference value.

11134 If *x*–*y* is positive and overflows, a range error shall occur and *fdim*( ), *fdimf*( ), and *fdiml*( ) shall
11135 return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

11136 XSI If *x*–*y* is positive and underflows, a range error may occur, and either (*x*–*y*) (if representable), or
11137 0.0 (if supported), or an implementation-defined value shall be returned.

11138 MX If *x* or *y* is NaN, a NaN shall be returned.

11139 **ERRORS**

11140 The *fdim*( ) function shall fail if:

11141 Range Error The result overflows.

11142 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
11143 then *errno* shall be set to [ERANGE]. If the integer expression |
11144 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow |
11145 floating-point exception shall be raised. |

11146 The *fdim*( ) function may fail if:

11147 Range Error The result underflows.

11148 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
11149 then *errno* shall be set to [ERANGE]. If the integer expression |
11150 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow |
11151 floating-point exception shall be raised. |

EXAMPLES

None.

APPLICATION USAGE

On implementations supporting IEEE Std 754-1985, $x-y$ cannot underflow, and hence the 0.0 return value is shaded as an extension for implementations supporting the XSI extension rather than an MX extension.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *fmax*( ), *fmin*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

11169 **NAME**

11170 fdopen — associate a stream with a file descriptor

11171 **SYNOPSIS**

11172 CX
```
#include <stdio.h>
```

11173
```
FILE *fdopen(int fildes, const char *mode);
```

11174

11175 **DESCRIPTION**

11176 The *fdopen*( ) function shall associate a stream with a file descriptor.

11177 The *mode* argument is a character string having one of the following values:

11178 *r* or *rb* Open a file for reading.

11179 *w* or *wb* Open a file for writing.

11180 *a* or *ab* Open a file for writing at end of file.

11181 *r+* or *rb+* or *r+b* Open a file for update (reading and writing).

11182 *w+* or *wb+* or *w+b* Open a file for update (reading and writing).

11183 *a+* or *ab+* or *a+b* Open a file for update (reading and writing) at end of file.

11184 The meaning of these flags is exactly as specified in *fopen*( ), except that modes beginning with *w*
11185 shall not cause truncation of the file.

11186 Additional values for the *mode* argument may be supported by an implementation.

11187 The application shall ensure that the mode of the stream as expressed by the *mode* argument is
11188 allowed by the file access mode of the open file description to which *fildes* refers. The file
11189 position indicator associated with the new stream is set to the position indicated by the file
11190 offset associated with the file descriptor.

11191 The error and end-of-file indicators for the stream shall be cleared. The *fdopen*( ) function may
11192 cause the *st_atime* field of the underlying file to be marked for update.

11193 SHM If *fildes* refers to a shared memory object, the result of the *fdopen*( ) function is unspecified.

11194 TYM If *fildes* refers to a typed memory object, the result of the *fdopen*( ) function is unspecified.

11195 The *fdopen*( ) function shall preserve the offset maximum previously set for the open file
11196 description corresponding to *fildes*.

11197 **RETURN VALUE**

11198 Upon successful completion, *fdopen*( ) shall return a pointer to a stream; otherwise, a null pointer
11199 shall be returned and *errno* set to indicate the error.

11200 **ERRORS**

11201 The *fdopen*( ) function may fail if:

11202 [EBADF] The *fildes* argument is not a valid file descriptor.

11203 [EINVAL] The *mode* argument is not a valid mode.

11204 [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

11205 [EMFILE] {STREAM_MAX} streams are currently open in the calling process.

11206 [ENOMEM] Insufficient space to allocate a buffer.

11207 **EXAMPLES**

11208 None.

11209 **APPLICATION USAGE**

11210 File descriptors are obtained from calls like *open*( ), *dup*( ), *creat*( ), or *pipe*( ), which open files but
11211 do not return streams.

11212 **RATIONALE**

11213 The file descriptor may have been obtained from *open*( ), *creat*( ), *pipe*( ), *dup*( ), or *fcntl*( );
11214 inherited through *fork*( ) or *exec*; or perhaps obtained by implementation-defined means, such as
11215 the 4.3 BSD *socket*( ) call.

11216 The meanings of the *mode* arguments of *fdopen*( ) and *fopen*( ) differ. With *fdopen*( ), open for write
11217 (*w* or *w+*) does not truncate, and append (*a* or *a+*) cannot create for writing. The *mode* argument
11218 formats that include *a b* are allowed for consistency with the ISO C standard function *fopen*( ).
11219 The *b* has no effect on the resulting stream. Although not explicitly required by this volume of
11220 IEEE Std 1003.1-200x, a good implementation of append (*a*) mode would cause the O_APPEND
11221 flag to be set.

11222 **FUTURE DIRECTIONS**

11223 None.

11224 **SEE ALSO**

11225 *fclose*( ), *fopen*( ), *open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, Section
11226 2.5.1 (on page 485)

11227 **CHANGE HISTORY**

11228 First released in Issue 1. Derived from Issue 1 of the SVID.

11229 **Issue 5**

11230 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

11231 Large File Summit extensions are added.

11232 **Issue 6**

11233 The following new requirements on POSIX implementations derive from alignment with the
11234 Single UNIX Specification:

11235 • In the DESCRIPTION, the use and setting of the *mode* argument are changed to include
11236 binary streams.

11237 • In the DESCRIPTION, text is added for large file support to indicate setting of the offset
11238 maximum in the open file description.

11239 • All errors identified in the ERRORS section are added.

11240 • In the DESCRIPTION, text is added that the *fdopen*( ) function may cause *st_atime* to be
11241 updated.

11242 The following changes were made to align with the IEEE P1003.1a draft standard:

11243 • Clarification is added that it is the responsibility of the application to ensure that the mode is
11244 compatible with the open file descriptor.

11245 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
11246 *fdopen*( ) results are unspecified for typed memory objects.

11247 **NAME**

11248 feclearexcept — clear floating-point exception

11249 **SYNOPSIS**

11250 `#include <fenv.h>`

11251 `int feclearexcept(int excepts);`

11252 **DESCRIPTION**

11253 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11254 conflict between the requirements described here and the ISO C standard is unintentional. This
11255 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11256 The *feclearexcept*() function shall attempt to clear the supported floating-point exceptions |
11257 represented by *excepts*. |

11258 **RETURN VALUE**

11259 If the argument is zero or if all the specified exceptions were successfully cleared, *feclearexcept*() |
11260 shall return zero. Otherwise, it shall return a non-zero value. |

11261 **ERRORS**

11262 No errors are defined.

11263 **EXAMPLES**

11264 None.

11265 **APPLICATION USAGE**

11266 None.

11267 **RATIONALE**

11268 None.

11269 **FUTURE DIRECTIONS**

11270 None.

11271 **SEE ALSO**

11272 *fegetexceptflag*(), *feraiseexcept*(), *fesetexceptflag*(), *fetestexcept*(), the Base Definitions volume of
11273 IEEE Std 1003.1-200x, **<fenv.h>**

11274 **CHANGE HISTORY**

11275 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard. |

11276 ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated. |

11277 **NAME**

11278 fegetenv, fesetenv — get and set current floating-point environment

11279 **SYNOPSIS**

11280 `#include <fenv.h>`

11281 `int fegetenv(fenv_t *envp);`
11282 `int fesetenv(const fenv_t *envp);`

11283 **DESCRIPTION**

11284 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11285 conflict between the requirements described here and the ISO C standard is unintentional. This
11286 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11287 The *fegetenv*( ) function shall attempt to store the current floating-point environment in the object |
11288 pointed to by *envp*. |

11289 The *fesetenv*( ) function shall attempt to establish the floating-point environment represented by |
11290 the object pointed to by *envp*. The argument *envp* shall point to an object set by a call to |
11291 *fegetenv*( ) or *feholdexcept*( ), or equal a floating-point environment macro. The *fesetenv*( ) function
11292 does not raise floating-point exceptions, but only installs the state of the floating-point status
11293 flags represented through its argument.

11294 **RETURN VALUE**

11295 If the representation was successfully stored, *fegetenv*( ) shall return zero. Otherwise, it shall |
11296 return a non-zero value. If the environment was successfully established, *fesetenv*( ) shall return |
11297 zero. Otherwise, it shall return a non-zero value. |

11298 **ERRORS**

11299 No errors are defined.

11300 **EXAMPLES**

11301 None.

11302 **APPLICATION USAGE**

11303 None.

11304 **RATIONALE**

11305 None.

11306 **FUTURE DIRECTIONS**

11307 None.

11308 **SEE ALSO**

11309 *feholdexcept*( ), *feupdateenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fenv.h>**

11310 **CHANGE HISTORY**

11311 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard. |

11312 ISO / IEC 9899: 1999 standard, Technical Corrigendum No. 1 is incorporated. |

11313 **NAME**

11314 fegetexceptflag, fesetexceptflag — get and set floating-point status flags

11315 **SYNOPSIS**

11316 `#include <fenv.h>`

11317 `int fegetexceptflag(fexcept_t *flagp, int excepts);`
11318 `int fesetexceptflag(const fexcept_t *flagp, int excepts);`

11319 **DESCRIPTION**

11320 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11321 conflict between the requirements described here and the ISO C standard is unintentional. This
11322 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11323 The *fegetexceptflag*( ) function shall attempt to store an implementation-defined representation of |
11324 the states of the floating-point status flags indicated by the argument *excepts* in the object |
11325 pointed to by the argument *flagp*.

11326 The *fesetexceptflag*( ) function shall attempt to set the floating-point status flags indicated by the |
11327 argument excepts to the states stored in the object pointed to by *flagp*. The value pointed to by |
11328 *flagp* shall have been set by a previous call to *fegetexceptflag*( ) whose second argument
11329 represented at least those floating-point exceptions represented by the argument *excepts*. This
11330 function does not raise floating-point exceptions, but only sets the state of the flags.

11331 **RETURN VALUE**

11332 If the representation was successfully stored, *fegetexceptflag*( ) shall return zero. Otherwise, it |
11333 shall return a non-zero value. If the *excepts* argument is zero or if all the specified execptions |
11334 were successfully set, *fesetexceptflag*( ) shall return zero. Otherwise, it shall return a non-zero |
11335 value. |

11336 **ERRORS**

11337 No errors are defined.

11338 **EXAMPLES**

11339 None.

11340 **APPLICATION USAGE**

11341 None.

11342 **RATIONALE**

11343 None.

11344 **FUTURE DIRECTIONS**

11345 None.

11346 **SEE ALSO**

11347 *feclearexcept*( ), *feraiseexcept*( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
11348 **<fenv.h>**

11349 **CHANGE HISTORY**

11350 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard. |

11351 ISO / IEC 9899: 1999 standard, Technical Corrigendum No. 1 is incorporated. |

11352 **NAME**

11353 fegetround, fesetround — get and set current rounding direction

11354 **SYNOPSIS**

11355 ```
#include <fenv.h>
```

11356 ```
int fegetround(void);
```

11357 ```
int fesetround(int round);
```

11358 **DESCRIPTION**

11359 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11360 conflict between the requirements described here and the ISO C standard is unintentional. This
11361 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11362 The *fegetround*( ) function shall get the current rounding direction.

11363 The *fesetround*( ) function shall establish the rounding direction represented by its argument
11364 *round*. If the argument is not equal to the value of a rounding direction macro, the rounding
11365 direction is not changed.

11366 **RETURN VALUE**

11367 The *fegetround*( ) function shall return the value of the rounding direction macro representing the
11368 current rounding direction or a negative value if there is no such rounding direction macro or
11369 the current rounding direction is not determinable.

11370 The *fesetround*( ) function shall return a zero value if and only if the requested rounding direction
11371 was established.

11372 **ERRORS**

11373 No errors are defined.

11374 **EXAMPLES**

11375 The following example saves, sets, and restores the rounding direction, reporting an error and
11376 aborting if setting the rounding direction fails:

```
11377 #include <fenv.h>
11378 #include <assert.h>
11379 void f(int round_dir)
11380 {
11381     #pragma STDC FENV_ACCESS ON
11382     int save_round;
11383     int setround_ok;
11384     save_round = fegetround();
11385     setround_ok = fesetround(round_dir);
11386     assert(setround_ok == 0);
11387     /* ... */
11388     fesetround(save_round);
11389     /* ... */
11390 }
```

11391 **APPLICATION USAGE**

11392 None.

11393 **RATIONALE**

11394 None.
```

11395 **FUTURE DIRECTIONS**

11396 None.

11397 **SEE ALSO**

11398 The Base Definitions volume of IEEE Std 1003.1-200x, **<fenv.h>**

11399 **CHANGE HISTORY**

11400 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard. |

11401 ISO / IEC 9899: 1999 standard, Technical Corrigendum No. 1 is incorporated. |

11402 **NAME**

11403 feholdexcept — save current floating-point environment

11404 **SYNOPSIS**

11405 `#include <fenv.h>`

11406 `int feholdexcept(fenv_t *envp);`

11407 **DESCRIPTION**

11408 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11409 conflict between the requirements described here and the ISO C standard is unintentional. This
11410 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11411 The *feholdexcept*() function shall save the current floating-point environment in the object
11412 pointed to by *envp*, clear the floating-point status flags, and then install a non-stop (continue on
11413 floating-point exceptions) mode, if available, for all floating-point exceptions.

11414 **RETURN VALUE**

11415 The *feholdexcept*() function shall return zero if and only if non-stop floating-point exception
11416 handling was successfully installed.

11417 **ERRORS**

11418 No errors are defined.

11419 **EXAMPLES**

11420 None.

11421 **APPLICATION USAGE**

11422 None.

11423 **RATIONALE**

11424 The *feholdexcept*() function should be effective on typical IEC 60559:1989 standard
11425 implementations which have the default non-stop mode and at least one other mode for trap
11426 handling or aborting. If the implementation provides only the non-stop mode, then installing the
11427 non-stop mode is trivial.

11428 **FUTURE DIRECTIONS**

11429 None.

11430 **SEE ALSO**

11431 *fegetenv*(), *fesetenv*(), *feupdateenv*(), the Base Definitions volume of IEEE Std 1003.1-200x,
11432 **<fenv.h>**

11433 **CHANGE HISTORY**

11434 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11435 **NAME**

11436 feof — test end-of-file indicator on a stream

11437 **SYNOPSIS**

11438 `#include <stdio.h>`

11439 `int feof(FILE *stream);`

11440 **DESCRIPTION**

11441 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11442 conflict between the requirements described here and the ISO C standard is unintentional. This
11443 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11444 The *feof*( ) function shall test the end-of-file indicator for the stream pointed to by *stream*.

11445 **RETURN VALUE**

11446 The *feof*( ) function shall return non-zero if and only if the end-of-file indicator is set for *stream*.

11447 **ERRORS**

11448 No errors are defined.

11449 **EXAMPLES**

11450 None.

11451 **APPLICATION USAGE**

11452 None.

11453 **RATIONALE**

11454 None.

11455 **FUTURE DIRECTIONS**

11456 None.

11457 **SEE ALSO**

11458 *clearerr*( ), *ferror*( ), *fopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

11459 **CHANGE HISTORY**

11460 First released in Issue 1. Derived from Issue 1 of the SVID.

11461 **NAME**

11462 feraiseexcept — raise floating-point exception

11463 **SYNOPSIS**

11464 `#include <fenv.h>`

11465 `int feraiseexcept(int excepts);`

11466 **DESCRIPTION**

11467 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11468 conflict between the requirements described here and the ISO C standard is unintentional. This
11469 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11470 The *feraiseexcept*() function shall attempt to raise the supported floating-point exceptions |
11471 represented by the argument *excepts*. The order in which these floating-point exceptions are
11472 raised is unspecified. Whether the *feraiseexcept*() function additionally raises the inexact
11473 floating-point exception whenever it raises the overflow or underflow floating-point exception is
11474 implementation-defined.

11475 **RETURN VALUE**

11476 If the argument is zero or if all the specified exceptions were successfully raised, *feraiseexcept*() |
11477 shall return zero. Otherwise, it shall return a non-zero value. |

11478 **ERRORS**

11479 No errors are defined.

11480 **EXAMPLES**

11481 None.

11482 **APPLICATION USAGE**

11483 The effect is intended to be similar to that of floating-point exceptions raised by arithmetic
11484 operations. Hence, enabled traps for floating-point exceptions raised by this function are taken.

11485 **RATIONALE**

11486 Raising overflow or underflow is allowed to also raise inexact because on some architectures the
11487 only practical way to raise an exception is to execute an instruction that has the exception as a
11488 side effect. The function is not restricted to accept only valid coincident expressions for atomic
11489 operations, so the function can be used to raise exceptions accrued over several operations.

11490 **FUTURE DIRECTIONS**

11491 None.

11492 **SEE ALSO**

11493 *feclearexcept*(), *fegetexceptflag*(), *fesetexceptflag*(), *fetestexcept*(), the Base Definitions volume of
11494 IEEE Std 1003.1-200x, **<fenv.h>**

11495 **CHANGE HISTORY**

11496 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard. |

11497 ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated. |

11498 **NAME**

11499 ferror — test error indicator on a stream

11500 **SYNOPSIS**

11501 `#include <stdio.h>`

11502 `int ferror(FILE *stream);`

11503 **DESCRIPTION**

11504 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11505 conflict between the requirements described here and the ISO C standard is unintentional. This
11506 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11507 The *ferror*( ) function shall test the error indicator for the stream pointed to by *stream*.

11508 **RETURN VALUE**

11509 The *ferror*( ) function shall return non-zero if and only if the error indicator is set for *stream*.

11510 **ERRORS**

11511 No errors are defined.

11512 **EXAMPLES**

11513 None.

11514 **APPLICATION USAGE**

11515 None.

11516 **RATIONALE**

11517 None.

11518 **FUTURE DIRECTIONS**

11519 None.

11520 **SEE ALSO**

11521 *clearerr*( ), *feof*( ), *fopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

11522 **CHANGE HISTORY**

11523 First released in Issue 1. Derived from Issue 1 of the SVID.

11524 **NAME**

11525 fesetenv — set current floating-point environment

11526 **SYNOPSIS**

11527 `#include <fenv.h>`

11528 `int fesetenv(const fenv_t *envp);`

11529 **DESCRIPTION**

11530 Refer to *fegetenv*( ).

11531 **NAME**

11532 fesetexceptflag — set floating-point status flags

11533 **SYNOPSIS**

11534 `#include <fenv.h>`

11535 `int fesetexceptflag(const fexcept_t *`*`flagp`*`, int `*`excepts`*`);`

11536 **DESCRIPTION**

11537 Refer to *fegetexceptflag*( ).

11538 **NAME**

11539 fesetround — set current rounding direction

11540 **SYNOPSIS**

11541 `#include <fenv.h>`

11542 `int fesetround(int round);`

11543 **DESCRIPTION**

11544 Refer to *fegetround*( ).

11545 **NAME**

11546 fetestexcept — test floating-point exception flags

11547 **SYNOPSIS**

11548 ```
#include <fenv.h>
```

11549 ```
int fetestexcept(int excepts);
```

11550 **DESCRIPTION**

11551 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11552 conflict between the requirements described here and the ISO C standard is unintentional. This
11553 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11554 The *fetestexcept*( ) function shall determine which of a specified subset of the floating-point
11555 exception flags are currently set. The *excepts* argument specifies the floating-point status flags to
11556 be queried.

11557 **RETURN VALUE**

11558 The *fetestexcept*( ) function shall return the value of the bitwise-inclusive OR of the floating-point
11559 exception macros corresponding to the currently set floating-point exceptions included in
11560 *excepts*.

11561 **ERRORS**

11562 No errors are defined.

11563 **EXAMPLES**

11564 The following example calls function *f*( ) if an invalid exception is set, and then function *g*( ) if an
11565 overflow exception is set:

```
11566 #include <fenv.h>
11567 /* ... */
11568 {
11569     #pragma STDC FENV_ACCESS ON
11570     int set_excepts;
11571     feclearexcept(FE_INVALID | FE_OVERFLOW);
11572     // maybe raise exceptions
11573     set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
11574     if (set_excepts & FE_INVALID) f();
11575     if (set_excepts & FE_OVERFLOW) g();
11576     /* ... */
11577 }
```

11578 **APPLICATION USAGE**

11579 None.

11580 **RATIONALE**

11581 None.

11582 **FUTURE DIRECTIONS**

11583 None.

11584 **SEE ALSO**

11585 *feclearexcept*( ), *fegetexceptflag*( ), *feraiseexcept*( ), the Base Definitions volume of
11586 IEEE Std 1003.1-200x, **<fenv.h>**

11587 **CHANGE HISTORY**
11588 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

11589 **NAME**

11590 feupdateenv — update floating-point environment

11591 **SYNOPSIS**

11592 ```
#include <fenv.h>
```

11593 ```
int feupdateenv(const fenv_t *envp);
```

11594 **DESCRIPTION**

11595 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11596 conflict between the requirements described here and the ISO C standard is unintentional. This
11597 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11598 The *feupdateenv*( ) function shall attempt to save the currently raised floating-point exceptions in |
11599 its automatic storage, attempt to install the floating-point environment represented by the object |
11600 pointed to by *envp*, and then attempt to raise the saved floating-point exceptions. The argument |
11601 *envp* shall point to an object set by a call to *feholdexcept*( ) or *fegetenv*( ), or equal a floating-point
11602 environment macro.

11603 **RETURN VALUE**

11604 The *feupdateenv*( ) function shall return a zero value if and only if all the required actions were |
11605 successfully carried out. |

11606 **ERRORS**

11607 No errors are defined.

11608 **EXAMPLES**

11609 The following example shows sample code to hide spurious underflow floating-point
11610 exceptions:

```
11611 #include <fenv.h>
11612 double f(double x)
11613 {
11614     #pragma STDC FENV_ACCESS ON
11615     double result;
11616     fenv_t save_env;
11617     feholdexcept(&save_env);
11618     // compute result
11619     if (/* test spurious underflow */)
11620     feclearexcept(FE_UNDERFLOW);
11621     feupdateenv(&save_env);
11622     return result;
11623 }
```

11624 **APPLICATION USAGE**

11625 None.

11626 **RATIONALE**

11627 None.

11628 **FUTURE DIRECTIONS**

11629 None.

11630 **SEE ALSO**

11631 *fegetenv*( ), *feholdexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fenv.h>**

11632 **CHANGE HISTORY**

11633 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard. |

11634 ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated. |

11635 **NAME**

11636 fflush — flush a stream

11637 **SYNOPSIS**

11638 `#include <stdio.h>`

11639 `int fflush(FILE *stream);`

11640 **DESCRIPTION**

11641 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11642 conflict between the requirements described here and the ISO C standard is unintentional. This
11643 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11644 If *stream* points to an output stream or an update stream in which the most recent operation was
11645 CX not input, *fflush*( ) shall cause any unwritten data for that stream to be written to the file, and the |
11646 *st_ctime* and *st_mtime* fields of the underlying file shall be marked for update. |

11647 If *stream* is a null pointer, *fflush*( ) shall perform this flushing action on all streams for which the
11648 behavior is defined above.

11649 **RETURN VALUE**

11650 Upon successful completion, *fflush*( ) shall return 0; otherwise, it shall set the error indicator for
11651 CX the stream, return EOF, and set *errno* to indicate the error.

11652 **ERRORS**

11653 The *fflush*( ) function shall fail if:

11654 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
11655 process would be delayed in the write operation.

11656 CX [EBADF] The file descriptor underlying *stream* is not valid.

11657 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

11658 XSI [EFBIG] An attempt was made to write a file that exceeds the process' file size limit.

11659 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
11660 offset maximum associated with the corresponding stream.

11661 CX [EINTR] The *fflush*( ) function was interrupted by a signal.

11662 CX [EIO] The process is a member of a background process group attempting to write
11663 to its controlling terminal, TOSTOP is set, the process is neither ignoring nor
11664 blocking SIGTTOU, and the process group of the process is orphaned. This
11665 error may also be returned under implementation-defined conditions.

11666 CX [ENOSPC] There was no free space remaining on the device containing the file.

11667 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
11668 any process. A SIGPIPE signal shall also be sent to the thread.

11669 The *fflush*( ) function may fail if:

11670 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
11671 capabilities of the device.

EXAMPLES

Sending Prompts to Standard Output

The following example uses *printf*() calls to print a series of prompts for information the user must enter from standard input. The *fflush*() calls force the output to standard output. The *fflush*() function is used because standard output is usually buffered and the prompt may not immediately be printed on the output or terminal. The *gets*() calls read strings from standard input and place the results in variables, for use later in the program

```
#include <stdio.h>
...
char user[100];
char oldpasswd[100];
char newpasswd[100];
...
printf("User name: ");
fflush(stdout);
gets(user);

printf("Old password: ");
fflush(stdout);
gets(oldpasswd);

printf("New password: ");
fflush(stdout);
gets(newpasswd);
...
```

APPLICATION USAGE

None.

RATIONALE

Data buffered by the system may make determining the validity of the position of the current file descriptor impractical. Thus, enforcing the repositioning of the file descriptor after *fflush*() on streams open for *read*() is not mandated by IEEE Std 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

getrlimit(), *ulimit*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EFBIG] error is added as part of the large file support extensions.
- The [ENXIO] optional error condition is added.

The RETURN VALUE section is updated to note that the error indicator shall be set for the stream. This is for alignment with the ISO/IEC 9899:1999 standard.

11717 **NAME**

11718 ffs — find first set bit

11719 **SYNOPSIS**

11720 XSI
```
#include <strings.h>
```

11721
```
int ffs(int i);
```

11722

11723 **DESCRIPTION**

11724 The *ffs*( ) function shall find the first bit set (beginning with the least significant bit) in *i*, and
11725 return the index of that bit. Bits are numbered starting at one (the least significant bit).

11726 **RETURN VALUE**

11727 The *ffs*( ) function shall return the index of the first bit set. If *i* is 0, then *ffs*( ) shall return 0.

11728 **ERRORS**

11729 No errors are defined.

11730 **EXAMPLES**

11731 None.

11732 **APPLICATION USAGE**

11733 None.

11734 **RATIONALE**

11735 None.

11736 **FUTURE DIRECTIONS**

11737 None.

11738 **SEE ALSO**

11739 The Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

11740 **CHANGE HISTORY**

11741 First released in Issue 4, Version 2.

11742 **Issue 5**

11743 Moved from X/OPEN UNIX extension to BASE.

11744 **NAME**

11745 fgetc — get a byte from a stream

11746 **SYNOPSIS**

11747 `#include <stdio.h>`

11748 `int fgetc(FILE *stream);`

11749 **DESCRIPTION**

11750 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11751 conflict between the requirements described here and the ISO C standard is unintentional. This
11752 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11753 If the end-of-file indicator for the input stream pointed to by *stream* is not set and a next byte is
11754 present, the *fgetc*( ) function shall obtain the next byte as an **unsigned char** converted to an **int**, |
11755 from the input stream pointed to by *stream*, and advance the associated file position indicator for |
11756 the stream (if defined). Since *fgetc*( ) operates on bytes, reading a character consisting of multiple |
11757 bytes (or ''a multi-byte character'') may require multiple calls to *fgetc*( ).

11758 CX The *fgetc*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The
11759 *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ),
11760 *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns
11761 data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

11762 **RETURN VALUE**

11763 Upon successful completion, *fgetc*( ) shall return the next byte from the input stream pointed to
11764 by *stream*. If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the
11765 end-of-file indicator for the stream shall be set and *fgetc*( ) shall return EOF. If a read error occurs,
11766 CX the error indicator for the stream shall be set, *fgetc*( ) shall return EOF, and shall set *errno* to
11767 indicate the error.

11768 **ERRORS**

11769 The *fgetc*( ) function shall fail if data needs to be read and:

11770 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
11771 process would be delayed in the *fgetc*( ) operation.

11772 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
11773 reading.

11774 CX [EINTR] The read operation was terminated due to the receipt of a signal, and no data
11775 was transferred.

11776 CX [EIO] A physical I/O error has occurred, or the process is in a background process
11777 group attempting to read from its controlling terminal, and either the process
11778 is ignoring or blocking the SIGTTIN signal or the process group is orphaned.
11779 This error may also be generated for implementation-defined reasons.

11780 CX [EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the
11781 offset maximum associated with the corresponding stream.

11782 The *fgetc*( ) function may fail if:

11783 CX [ENOMEM] Insufficient storage space is available.

11784 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
11785 capabilities of the device.

EXAMPLES

None.

APPLICATION USAGE

If the integer value returned by *fgetc*() is stored into a variable of type **char** and then compared against the integer constant EOF, the comparison may never succeed, because sign-extension of a variable of type **char** on widening to integer is implementation-defined.

The *ferror*() or *feof*() functions must be used to distinguish between an error condition and an end-of-file condition.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feof(), *ferror*(), *fopen*(), *getchar*(), *getc*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] and [EOVERFLOW] mandatory error conditions are added.
- The [ENOMEM] and [ENXIO] optional error conditions are added.

The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- The DESCRIPTION is updated to clarify the behavior when the end-of-file indicator for the input stream is not set.
- The RETURN VALUE section is updated to note that the error indicator shall be set for the stream.

11816 **NAME**

11817 fgetpos — get current file position information

11818 **SYNOPSIS**

11819 `#include <stdio.h>`

11820 `int fgetpos(FILE *restrict stream, fpos_t *restrict pos);`

11821 **DESCRIPTION**

11822 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11823 conflict between the requirements described here and the ISO C standard is unintentional. This
11824 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11825 The *fgetpos*( ) function shall store the current values of the parse state (if any) and file position
11826 indicator for the stream pointed to by *stream* in the object pointed to by *pos*. The value stored
11827 contains unspecified information usable by *fsetpos*( ) for repositioning the stream to its position
11828 at the time of the call to *fgetpos*( ).

11829 **RETURN VALUE**

11830 Upon successful completion, *fgetpos*( ) shall return 0; otherwise, it shall return a non-zero value
11831 and set *errno* to indicate the error.

11832 **ERRORS**

11833 The *fgetpos*( ) function shall fail if:

11834 CX [EOVERFLOW] The current value of the file position cannot be represented correctly in an
11835 object of type **fpos_t**.

11836 The *fgetpos*( ) function may fail if:

11837 CX [EBADF] The file descriptor underlying *stream* is not valid.

11838 CX [ESPIPE] The file descriptor underlying *stream* is associated with a pipe, FIFO, or socket.

11839

11840 **EXAMPLES**

11841 None.

11842 **APPLICATION USAGE**

11843 None.

11844 **RATIONALE**

11845 None.

11846 **FUTURE DIRECTIONS**

11847 None.

11848 **SEE ALSO**

11849 *fopen*( ), *ftell*( ), *rewind*( ), *ungetc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

11850 **CHANGE HISTORY**

11851 First released in Issue 4. Derived from the ISO C standard.

11852 **Issue 5**

11853 Large File Summit extensions are added.

11854 **Issue 6**

11855 Extensions beyond the ISO C standard are now marked.

11856 The following new requirements on POSIX implementations derive from alignment with the
11857 Single UNIX Specification:

- The [EIO] mandatory error condition is added.
- The [EBADF] and [ESPIPE] optional error conditions are added.

An additional [ESPIPE] error condition is added for sockets.

The prototype for *fgetpos*( ) is changed for alignment with the ISO / IEC 9899: 1999 standard.

11862 **NAME**

11863 fgets — get a string from a stream

11864 **SYNOPSIS**

11865 `#include <stdio.h>`

11866 `char *fgets(char *restrict s, int n, FILE *restrict stream);`

11867 **DESCRIPTION**

11868 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11869 conflict between the requirements described here and the ISO C standard is unintentional. This
11870 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11871 The *fgets*( ) function shall read bytes from *stream* into the array pointed to by *s*, until *n*−1 bytes
11872 are read, or a <newline> is read and transferred to *s*, or an end-of-file condition is encountered.
11873 The string is then terminated with a null byte.

11874 CX The *fgets*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The
11875 *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ),
11876 *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns
11877 data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

11878 **RETURN VALUE**

11879 Upon successful completion, *fgets*( ) shall return *s*. If the stream is at end-of-file, the end-of-file
11880 indicator for the stream shall be set and *fgets*( ) shall return a null pointer. If a read error occurs,
11881 CX the error indicator for the stream shall be set, *fgets*( ) shall return a null pointer, and shall set
11882 *errno* to indicate the error.

11883 **ERRORS**

11884 Refer to *fgetc*( ).

11885 **EXAMPLES**

11886 **Reading Input**

11887 The following example uses *fgets*( ) to read each line of input. {LINE_MAX}, which defines the
11888 maximum size of the input line, is defined in the **<limits.h>** header.

```
11889 #include <stdio.h>
11890 ...
11891 char line[LINE_MAX];
11892 ...
11893 while (fgets(line, LINE_MAX, fp) != NULL) {
11894 ...
11895 }
11896 ...
```

11897 **APPLICATION USAGE**

11898 None.

11899 **RATIONALE**

11900 None.

11901 **FUTURE DIRECTIONS**

11902 None.

11903 **SEE ALSO**

11904 *fopen*( ), *fread*( ), *gets*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

11905 **CHANGE HISTORY**

11906 First released in Issue 1. Derived from Issue 1 of the SVID.

11907 **Issue 6**

11908 Extensions beyond the ISO C standard are now marked.

11909 The prototype for *fgets*( ) is changed for alignment with the ISO / IEC 9899: 1999 standard.

11910 **NAME**

11911 fgetwc — get a wide-character code from a stream

11912 **SYNOPSIS**

11913 ```
#include <stdio.h>
```

11914 ```
#include <wchar.h>
```

11915 ```
wint_t fgetwc(FILE *stream);
```

11916 **DESCRIPTION**

11917 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11918 conflict between the requirements described here and the ISO C standard is unintentional. This
11919 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11920 The *fgetwc*( ) function shall obtain the next character (if present) from the input stream pointed to
11921 by *stream*, convert that to the corresponding wide-character code, and advance the associated
11922 file position indicator for the stream (if defined).

11923 If an error occurs, the resulting value of the file position indicator for the stream is unspecified. |

11924 CX The *fgetwc*( ) function may mark the *st_atime* field of the file associated with *stream* for update.
11925 The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ),
11926 *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns
11927 data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

11928 **RETURN VALUE**

11929 Upon successful completion, the *fgetwc*( ) function shall return the wide-character code of the
11930 character read from the input stream pointed to by *stream* converted to a type **wint_t**. If the
11931 stream is at end-of-file, the end-of-file indicator for the stream shall be set and *fgetwc*( ) shall
11932 return WEOF. If a read error occurs, the error indicator for the stream shall be set, *fgetwc*( ) shall
11933 CX return WEOF, and shall set *errno* to indicate the error. If an encoding error occurs, the error |
11934 indicator for the stream shall be set, *fgetwc*( ) shall return WEOF, and shall set *errno* to indicate |
11935 the error. |

11936 **ERRORS**

11937 The *fgetwc*( ) function shall fail if data needs to be read and:

11938 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
11939 process would be delayed in the *fgetwc*( ) operation.

11940 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
11941 reading. |

11942 [EILSEQ] The data obtained from the input stream does not form a valid character. |

11943 CX [EINTR] The read operation was terminated due to the receipt of a signal, and no data
11944 was transferred.

11945 CX [EIO] A physical I/O error has occurred, or the process is in a background process
11946 group attempting to read from its controlling terminal, and either the process
11947 is ignoring or blocking the SIGTTIN signal or the process group is orphaned.
11948 This error may also be generated for implementation-defined reasons.

11949 CX [EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the
11950 offset maximum associated with the corresponding stream.

11951 The *fgetwc*( ) function may fail if:

11952 CX [ENOMEM] Insufficient storage space is available.
```

11953 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
11954 capabilities of the device. |

11955 **EXAMPLES**

11956 None.

11957 **APPLICATION USAGE**

11958 The *ferror*( ) or *feof*( ) functions must be used to distinguish between an error condition and an
11959 end-of-file condition.

11960 **RATIONALE**

11961 None.

11962 **FUTURE DIRECTIONS**

11963 None.

11964 **SEE ALSO**

11965 *feof*( ), *ferror*( ), *fopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**,
11966 **<wchar.h>**

11967 **CHANGE HISTORY**

11968 First released in Issue 4. Derived from the MSE working draft.

11969 **Issue 5**

11970 The Optional Header (OH) marking is removed from **<stdio.h>**.

11971 Large File Summit extensions are added.

11972 **Issue 6**

11973 Extensions beyond the ISO C standard are now marked.

11974 The following new requirements on POSIX implementations derive from alignment with the
11975 Single UNIX Specification:

11976 - The [EIO] and [EOVERFLOW] mandatory error conditions are added.

11977 - The [ENOMEM] and [ENXIO] optional error conditions are added. |

11978 **NAME**

11979 fgetws — get a wide-character string from a stream

11980 **SYNOPSIS**

11981 ```
#include <stdio.h>
```

11982 ```
#include <wchar.h>
```

11983 ```
wchar_t *fgetws(wchar_t *restrict ws, int n,
```

11984 ```
    FILE *restrict stream);
```

11985 **DESCRIPTION**

11986 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11987 conflict between the requirements described here and the ISO C standard is unintentional. This
11988 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11989 The *fgetws*( ) function shall read characters from the *stream*, convert these to the corresponding
11990 wide-character codes, place them in the **wchar_t** array pointed to by *ws*, until *n*−1 characters are
11991 read, or a <newline> is read, converted, and transferred to *ws*, or an end-of-file condition is
11992 encountered. The wide-character string, *ws*, shall then be terminated with a null wide-character |
11993 code. |

11994 If an error occurs, the resulting value of the file position indicator for the stream is unspecified. |

11995 CX The *fgetws*( ) function may mark the *st_atime* field of the file associated with *stream* for update.
11996 The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ),
11997 *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns
11998 data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

11999 **RETURN VALUE**

12000 Upon successful completion, *fgetws*( ) shall return *ws*. If the stream is at end-of-file, the end-of-
12001 file indicator for the stream shall be set and *fgetws*( ) shall return a null pointer. If a read error
12002 CX occurs, the error indicator for the stream shall be set, *fgetws*( ) shall return a null pointer, and
12003 shall set *errno* to indicate the error.

12004 **ERRORS**

12005 Refer to *fgetwc*( ).

12006 **EXAMPLES**

12007 None.

12008 **APPLICATION USAGE**

12009 None.

12010 **RATIONALE**

12011 None.

12012 **FUTURE DIRECTIONS**

12013 None.

12014 **SEE ALSO**

12015 *fopen*( ), *fread*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

12016 **CHANGE HISTORY**

12017 First released in Issue 4. Derived from the MSE working draft.

12018 **Issue 5**

12019 The Optional Header (OH) marking is removed from **<stdio.h>**.

12020 **Issue 6**

12021 Extensions beyond the ISO C standard are now marked.

12022 The prototype for *fgetws*( ) is changed for alignment with the ISO / IEC 9899: 1999 standard.

12023 **NAME**

12024 fileno — map a stream pointer to a file descriptor

12025 **SYNOPSIS**

12026 CX
```
#include <stdio.h>

int fileno(FILE *stream);
```
12027

12028

12029 **DESCRIPTION**

12030 The *fileno*( ) function shall return the integer file descriptor associated with the stream pointed to
12031 by *stream*.

12032 **RETURN VALUE**

12033 Upon successful completion, *fileno*( ) shall return the integer value of the file descriptor
12034 associated with *stream*. Otherwise, the value –1 shall be returned and *errno* set to indicate the
12035 error.

12036 **ERRORS**

12037 The *fileno*( ) function may fail if:

12038 [EBADF] The *stream* argument is not a valid stream.

12039 **EXAMPLES**

12040 None.

12041 **APPLICATION USAGE**

12042 None.

12043 **RATIONALE**

12044 Without some specification of which file descriptors are associated with these streams, it is
12045 impossible for an application to set up the streams for another application it starts with *fork*( )
12046 and *exec*. In particular, it would not be possible to write a portable version of the *sh* command
12047 interpreter (although there may be other constraints that would prevent that portability).

12048 **FUTURE DIRECTIONS**

12049 None.

12050 **SEE ALSO**

12051 *fdopen*( ), *fopen*( ), *stdin*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, Section
12052 2.5.1 (on page 485)

12053 **CHANGE HISTORY**

12054 First released in Issue 1. Derived from Issue 1 of the SVID.

12055 **Issue 6**

12056 The following new requirements on POSIX implementations derive from alignment with the
12057 Single UNIX Specification:

12058 • The [EBADF] optional error condition is added.

12059 **NAME**

12060 flockfile, ftrylockfile, funlockfile — stdio locking functions

12061 **SYNOPSIS**

12062 TSF
```
#include <stdio.h>

void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);
```
12066

12067 **DESCRIPTION**

12068 These functions shall provide for explicit application-level locking of stdio (**FILE ***) objects. |
12069 These functions can be used by a thread to delineate a sequence of I/O statements that are
12070 executed as a unit.

12071 The *flockfile*( ) function shall acquire for a thread ownership of a (**FILE ***) object. |

12072 The *ftrylockfile*( ) function shall acquire for a thread ownership of a (**FILE ***) object if the object is |
12073 available; *ftrylockfile*( ) is a non-blocking version of *flockfile*( ).

12074 The *funlockfile*( ) function shall relinquish the ownership granted to the thread. The behavior is |
12075 undefined if a thread other than the current owner calls the *funlockfile*( ) function.

12076 The functions shall behave as if there is a lock count associated with each (**FILE ***) object. This |
12077 count is implicitly initialized to zero when the (**FILE ***) object is created. The (**FILE ***) object is
12078 unlocked when the count is zero. When the count is positive, a single thread owns the (**FILE ***)
12079 object. When the *flockfile*( ) function is called, if the count is zero or if the count is positive and
12080 the caller owns the (**FILE ***) object, the count shall be incremented. Otherwise, the calling thread |
12081 shall be suspended, waiting for the count to return to zero. Each call to *funlockfile*( ) shall |
12082 decrement the count. This allows matching calls to *flockfile*( ) (or successful calls to *ftrylockfile*( )) |
12083 and *funlockfile*( ) to be nested.

12084 All functions that reference (**FILE ***) objects shall behave as if they use *flockfile*( ) and *funlockfile*( )
12085 internally to obtain ownership of these (**FILE ***) objects.

12086 **RETURN VALUE**

12087 None for *flockfile*( ) and *funlockfile*( ). The *ftrylockfile*( ) function shall return zero for success and
12088 non-zero to indicate that the lock cannot be acquired.

12089 **ERRORS**

12090 No errors are defined.

12091 **EXAMPLES**

12092 None.

12093 **APPLICATION USAGE**

12094 Applications using these functions may be subject to priority inversion, as discussed in the Base
12095 Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

12096 **RATIONALE**

12097 The *flockfile*( ) and *funlockfile*( ) functions provide an orthogonal mutual exclusion lock for each
12098 **FILE**. The *ftrylockfile*( ) function provides a non-blocking attempt to acquire a file lock,
12099 analogous to *pthread_mutex_trylock*( ).

12100 These locks behave as if they are the same as those used internally by *stdio* for thread-safety.
12101 This both provides thread-safety of these functions without requiring a second level of internal
12102 locking and allows functions in *stdio* to be implemented in terms of other *stdio* functions.

Application writers and implementors should be aware that there are potential deadlock problems on **FILE** objects. For example, the line-buffered flushing semantics of *stdio* (requested via {_IOLBF}) require that certain input operations sometimes cause the buffered contents of implementation-defined line-buffered output streams to be flushed. If two threads each hold the lock on the other's **FILE**, deadlock ensues. This type of deadlock can be avoided by acquiring **FILE** locks in a consistent order. In particular, the line-buffered output stream deadlock can typically be avoided by acquiring locks on input streams before locks on output streams if a thread would be acquiring both.

In summary, threads sharing *stdio* streams with other threads can use *flockfile*( ) and *funlockfile*( ) to cause sequences of I/O performed by a single thread to be kept bundled. The only case where the use of *flockfile*( ) and *funlockfile*( ) is required is to provide a scope protecting uses of the **_unlocked*( ) functions/macros. This moves the cost/performance tradeoff to the optimal point.

FUTURE DIRECTIONS

None.

SEE ALSO

getc_unlocked( ), *putc_unlocked*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

These functions are marked as part of the Thread-Safe Functions option.

12123 **NAME**

12124 floor, floorf, floorl — floor function

12125 **SYNOPSIS**

12126 `#include <math.h>`

12127 `double floor(double x);`
12128 `float floorf(float x);`
12129 `long double floorl(long double x);`

12130 **DESCRIPTION**

12131 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12132 conflict between the requirements described here and the ISO C standard is unintentional. This
12133 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12134 These functions shall compute the largest integral value not greater than *x*.

12135 An application wishing to check for error situations should set *errno* to zero and call
12136 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
12137 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
12138 zero, an error has occurred.

12139 **RETURN VALUE**

12140 Upon successful completion, these functions shall return the largest integral value not greater
12141 than *x*, expressed as a **double**, **float**, or **long double**, as appropriate for the return type of the
12142 function.

12143 MX If *x* is NaN, a NaN shall be returned.

12144 If *x* is ±0 or ±Inf, *x* shall be returned.

12145 XSI If the correct value would cause overflow, a range error shall occur and *floor*(), *floorf*(), and
12146 *floorl*() shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
12147 respectively.

12148 **ERRORS**

12149 These functions shall fail if:

12150 XSI

Range Error	The result would cause an overflow.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

12155 **EXAMPLES**

12156 None.

12157 **APPLICATION USAGE**

12158 The integral value returned by these functions might not be expressible as an **int** or **long**. The
12159 return value should be tested before assigning it to an integer type to avoid the undefined results
12160 of an integer overflow.

12161 The *floor*() function can only overflow when the floating-point representation has
12162 DBL_MANT_DIG > DBL_MAX_EXP.

12163 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
12164 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

12165 **RATIONALE**

12166 None.

12167 **FUTURE DIRECTIONS**

12168 None.

12169 **SEE ALSO**

12170 *ceil*( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, |
12171 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

12172 **CHANGE HISTORY**

12173 First released in Issue 1. Derived from Issue 1 of the SVID.

12174 **Issue 5**

12175 The DESCRIPTION is updated to indicate how an application should check for an error. This
12176 text was previously published in the APPLICATION USAGE section.

12177 **Issue 6**

12178 The *floorf*( ) and *floorl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

12179 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
12180 revised to align with the ISO / IEC 9899: 1999 standard.

12181 IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are
12182 marked.

12183 **NAME**

12184 fma, fmaf, fmal — floating-point multiply-add

12185 **SYNOPSIS**

12186
```
#include <math.h>
```

12187
```
double fma(double x, double y, double z);
```
12188
```
float fmaf(float x, float y, float z);
```
12189
```
long double fmal(long double x, long double y, long double z);
```

12190 **DESCRIPTION**

12191 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12192 conflict between the requirements described here and the ISO C standard is unintentional. This
12193 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12194 These functions shall compute $(x * y) + z$, rounded as one ternary operation: they shall compute
12195 the value (as if) to infinite precision and round once to the result format, according to the
12196 rounding mode characterized by the value of FLT_ROUNDS.

12197 An application wishing to check for error situations should set *errno* to zero and call
12198 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
12199 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
12200 zero, an error has occurred.

12201 **RETURN VALUE**

12202 Upon successful completion, these functions shall return $(x * y) + z$, rounded as one ternary
12203 operation.

12204 MX If x or y are NaN, a NaN shall be returned.

12205 If x multiplied by y is an exact infinity and z is also an infinity but with the opposite sign, a
12206 domain error shall occur, and either a NaN (if supported), or an implementation-defined value
12207 shall be returned.

12208 If one of x and y is infinite, the other is zero, and z is not a NaN, a domain error shall occur, and
12209 either a NaN (if supported), or an implementation-defined value shall be returned.

12210 If one of x and y is infinite, the other is zero, and z is a NaN, a NaN shall be returned and a
12211 domain error may occur.

12212 If $x*y$ is not 0*Inf nor Inf*0 and z is a NaN, a NaN shall be returned.

12213 **ERRORS**

12214 These functions shall fail if:

12215 MX

Domain Error	The value of $x*y+z$ is invalid, or the value $x*y$ is invalid and z is not a NaN.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
Range Error	The result overflows.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

12225 These functions may fail if:

MX	Domain Error	The value $x*y$ is invalid and z is a NaN.	
		If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.	
MX	Range Error	The result underflows.	
		If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.	

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

In many cases, clever use of floating (*fused*) multiply-add leads to much improved code; but its unexpected use by the compiler can undermine carefully written code. The FP_CONTRACT macro can be used to disallow use of floating multiply-add; and the *fma*( ) function guarantees its use where desired. Many current machines provide hardware floating multiply-add instructions; software implementation can be used for others.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

12254 **NAME**

12255 fmax, fmaxf, fmaxl — determine maximum numeric value of two floating-point numbers

12256 **SYNOPSIS**

12257
```
#include <math.h>
```

12258
```
double fmax(double x, double y);
```
12259
```
float fmaxf(float x, float y);
```
12260
```
long double fmaxl(long double x, long double y);
```

12261 **DESCRIPTION**

12262 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12263 conflict between the requirements described here and the ISO C standard is unintentional. This
12264 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12265 These functions shall determine the maximum numeric value of their arguments. NaN
12266 arguments shall be treated as missing data: if one argument is a NaN and the other numeric,
12267 then these functions shall choose the numeric value.

12268 **RETURN VALUE**

12269 Upon successful completion, these functions shall return the maximum numeric value of their
12270 arguments.

12271 If just one argument is a NaN, the other argument shall be returned.

12272 MX If *x* and *y* are NaN, a NaN shall be returned.

12273 **ERRORS**

12274 No errors are defined.

12275 **EXAMPLES**

12276 None.

12277 **APPLICATION USAGE**

12278 None.

12279 **RATIONALE**

12280 None.

12281 **FUTURE DIRECTIONS**

12282 None.

12283 **SEE ALSO**

12284 *fdim*( ), *fmin*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

12285 **CHANGE HISTORY**

12286 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

12287 **NAME**

12288 fmin, fminf, fminl — determine minimum numeric value of two floating-point numbers

12289 **SYNOPSIS**

12290
```
#include <math.h>
```

12291
```
double fmin(double x, double y);
```
12292
```
float fminf(float x, float y);
```
12293
```
long double fminl(long double x, long double y);
```

12294 **DESCRIPTION**

12295 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12296 conflict between the requirements described here and the ISO C standard is unintentional. This
12297 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12298 These functions shall determine the minimum numeric value of their arguments. NaN
12299 arguments shall be treated as missing data: if one argument is a NaN and the other numeric,
12300 then these functions shall choose the numeric value.

12301 **RETURN VALUE**

12302 Upon successful completion, these functions shall return the minimum numeric value of their
12303 arguments.

12304 If just one argument is a NaN, the other argument shall be returned.

12305 MX If *x* and *y* are NaN, a NaN shall be returned.

12306 **ERRORS**

12307 No errors are defined.

12308 **EXAMPLES**

12309 None.

12310 **APPLICATION USAGE**

12311 None.

12312 **RATIONALE**

12313 None.

12314 **FUTURE DIRECTIONS**

12315 None.

12316 **SEE ALSO**

12317 *fdim*( ), *fmax*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

12318 **CHANGE HISTORY**

12319 First released in Issue 6. Derived from ISO / IEC 9899: 1999 standard.

12320 **NAME**

12321 fmod, fmodf, fmodl — floating-point remainder value function

12322 **SYNOPSIS**

12323
```
#include <math.h>
```

12324
```
double fmod(double x, double y);
```
12325
```
float fmodf(float x, float y);
```
12326
```
long double fmodl(long double x, long double y);
```

12327 **DESCRIPTION**

12328 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12329 conflict between the requirements described here and the ISO C standard is unintentional. This
12330 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12331 These functions shall return the floating-point remainder of the division of *x* by *y*.

12332 An application wishing to check for error situations should set *errno* to zero and call
12333 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
12334 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
12335 zero, an error has occurred.

12336 **RETURN VALUE**

12337 These functions shall return the value $x-i*y$, for some integer i such that, if y is non-zero, the
12338 result has the same sign as x and magnitude less than the magnitude of y.

12339 If the correct value would cause underflow, and is not representable, a range error may occur,
12340 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

12341 MX If *x* or *y* is NaN, a NaN shall be returned.

12342 If *y* is zero, a domain error shall occur, and either a NaN (if supported), or an implementation-
12343 defined value shall be returned.

12344 If *x* is infinite, a domain error shall occur, and either a NaN (if supported), or an
12345 implementation-defined value shall be returned.

12346 If *x* is ±0 and *y* is not zero, ±0 shall be returned.

12347 If *x* is not infinite and *y* is ±Inf, *x* shall be returned.

12348 If the correct value would cause underflow, and is representable, a range error may occur and
12349 the correct value shall be returned.

12350 **ERRORS**

12351 These functions shall fail if:

12352 MX Domain Error The *x* argument is infinite or *y* is zero.

12353 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
12354 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling |
12355 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception |
12356 shall be raised. |

12357 These functions may fail if:

12358 Range Error The result underflows.

12359 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
12360 then *errno* shall be set to [ERANGE]. If the integer expression |
12361 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow |
12362 floating-point exception shall be raised. |

12363 **EXAMPLES**

12364 None.

12365 **APPLICATION USAGE**

12366 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
12367 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

12368 **RATIONALE**

12369 None.

12370 **FUTURE DIRECTIONS**

12371 None.

12372 **SEE ALSO**

12373 *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, |
12374 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

12375 **CHANGE HISTORY**

12376 First released in Issue 1. Derived from Issue 1 of the SVID.

12377 **Issue 5**

12378 The DESCRIPTION is updated to indicate how an application should check for an error. This
12379 text was previously published in the APPLICATION USAGE section.

12380 **Issue 6**

12381 The behavior for when the *y* argument is zero is now defined.

12382 The *fmodf*( ) and *fmodl*( ) functions are added for alignment with the ISO / IEC 9899: 1999
12383 standard.

12384 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
12385 revised to align with the ISO / IEC 9899: 1999 standard.

12386 IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are
12387 marked.

12388 **NAME**

12389 fmtmsg — display a message in the specified format on standard error and/or a system console

12390 **SYNOPSIS**

12391 XSI
```
#include <fmtmsg.h>

int fmtmsg(long classification, const char *label, int severity,
    const char *text, const char *action, const char *tag);
```
12394

12395 **DESCRIPTION**

12396 The *fmtmsg*( ) function shall display messages in a specified format instead of the traditional
12397 *printf*( ) function.

12398 Based on a message's classification component, *fmtmsg*( ) shall write a formatted message either
12399 to standard error, to the console, or to both.

12400 A formatted message consists of up to five components as defined below. The component
12401 *classification* is not part of a message displayed to the user, but defines the source of the message
12402 and directs the display of the formatted message.

12403 *classification* Contains the sum of identifying values constructed from the constants defined
12404 below. Any one identifier from a subclass may be used in combination with a
12405 single identifier from a different subclass. Two or more identifiers from the
12406 same subclass should not be used together, with the exception of identifiers
12407 from the display subclass. (Both display subclass identifiers may be used so
12408 that messages can be displayed to both standard error and the system
12409 console).

12410 **Major Classifications**

12411 Identifies the source of the condition. Identifiers are: MM_HARD
12412 (hardware), MM_SOFT (software), and MM_FIRM (firmware).

12413 **Message Source Subclassifications**

12414 Identifies the type of software in which the problem is detected.
12415 Identifiers are: MM_APPL (application), MM_UTIL (utility), and
12416 MM_OPSYS (operating system).

12417 **Display Subclassifications**

12418 Indicates where the message is to be displayed. Identifiers are:
12419 MM_PRINT to display the message on the standard error stream,
12420 MM_CONSOLE to display the message on the system console. One or
12421 both identifiers may be used.

12422 **Status Subclassifications**

12423 Indicates whether the application can recover from the condition.
12424 Identifiers are: MM_RECOVER (recoverable) and MM_NRECOV (non-
12425 recoverable).

12426 An additional identifier, MM_NULLMC, indicates that no classification
12427 component is supplied for the message.

12428 *label* Identifies the source of the message. The format is two fields separated by a
12429 colon. The first field is up to 10 bytes, the second is up to 14 bytes.

12430 *severity* Indicates the seriousness of the condition. Identifiers for the levels of *severity*
12431 are:

MM_HALT	Indicates that the application has encountered a severe fault and is halting. Produces the string "HALT".
MM_ERROR	Indicates that the application has detected a fault. Produces the string "ERROR".
MM_WARNING	Indicates a condition that is out of the ordinary, that might be a problem, and should be watched. Produces the string "WARNING".
MM_INFO	Provides information about a condition that is not in error. Produces the string "INFO".
MM_NOSEV	Indicates that no severity level is supplied for the message.

text Describes the error condition that produced the message. The character string is not limited to a specific size. If the character string is empty, then the text produced is unspecified.

action Describes the first step to be taken in the error-recovery process. The *fmtmsg*( ) function precedes the action string with the prefix: "TO FIX:". The *action* string is not limited to a specific size.

tag An identifier that references on-line documentation for the message. Suggested usage is that *tag* includes the *label* and a unique identifying number. A sample *tag* is "XSI:cat:146".

The *MSGVERB* environment variable (for message verbosity) shall determine for *fmtmsg*( ) which message components it is to select when writing messages to standard error. The value of *MSGVERB* shall be a colon-separated list of optional keywords. Valid *keywords* are: *label*, *severity*, *text*, *action*, and *tag*. If *MSGVERB* contains a keyword for a component and the component's value is not the component's null value, *fmtmsg*( ) shall include that component in the message when writing the message to standard error. If *MSGVERB* does not include a keyword for a message component, that component is shall not be included in the display of the message. The keywords may appear in any order. If *MSGVERB* is not defined, if its value is the null string, if its value is not of the correct format, or if it contains keywords other than the valid ones listed above, *fmtmsg*( ) shall select all components.

MSGVERB shall determine which components are selected for display to standard error. All message components shall be included in console messages.

RETURN VALUE

The *fmtmsg*( ) function shall return one of the following values:

MM_OK	The function succeeded.
MM_NOTOK	The function failed completely.
MM_NOMSG	The function was unable to generate a message on standard error, but otherwise succeeded.
MM_NOCON	The function was unable to generate a console message, but otherwise succeeded.

ERRORS

None.

12473 **EXAMPLES**

12474 1. The following example of *fmtmsg*( ):

12475 ```
12476 fmtmsg(MM_PRINT, "XSI:cat", MM_ERROR, "illegal option",
      "refer to cat in user's reference manual", "XSI:cat:001")
```

12477 produces a complete message in the specified message format:

12478 ```
12479 XSI:cat: ERROR: illegal option
      TO FIX: refer to cat in user's reference manual XSI:cat:001
```

12480 2. When the environment variable *MSGVERB* is set as follows:

12481 ```
      MSGVERB=severity:text:action
```

12482 and Example 1 is used, *fmtmsg*( ) produces:

12483 ```
12484 ERROR: illegal option
      TO FIX: refer to cat in user's reference manual
```

12485 **APPLICATION USAGE**

12486 One or more message components may be systematically omitted from messages generated by
12487 an application by using the null value of the argument for that component.

12488 **RATIONALE**

12489 None.

12490 **FUTURE DIRECTIONS**

12491 None.

12492 **SEE ALSO**

12493 *printf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fmtmsg.h>**

12494 **CHANGE HISTORY**

12495 First released in Issue 4, Version 2.

12496 **Issue 5**

12497 Moved from X/OPEN UNIX extension to BASE.

12498 **NAME**

12499 fnmatch — match a filename or a pathname

12500 **SYNOPSIS**

12501 `#include <fnmatch.h>`

12502 `int fnmatch(const char *pattern, const char *string, int flags);`

12503 **DESCRIPTION**

12504 The *fnmatch*( ) function shall match patterns as described in the Shell and Utilities volume of
12505 IEEE Std 1003.1-200x, Section 2.13.1, Patterns Matching a Single Character, and Section 2.13.2,
12506 Patterns Matching Multiple Characters. It checks the string specified by the *string* argument to
12507 see if it matches the pattern specified by the *pattern* argument.

12508 The *flags* argument shall modify the interpretation of *pattern* and *string*. It is the bitwise-inclusive
12509 OR of zero or more of the flags defined in **<fnmatch.h>**. If the FNM_PATHNAME flag is set in
12510 *flags*, then a slash character (`'/'`) in *string* shall be explicitly matched by a slash in *pattern*; it shall
12511 not be matched by either the asterisk or question-mark special characters, nor by a bracket
12512 expression. If the FNM_PATHNAME flag is not set, the slash character shall be treated as an
12513 ordinary character.

12514 If FNM_NOESCAPE is not set in *flags*, a backslash character (`'\'`) in *pattern* followed by any
12515 other character shall match that second character in *string*. In particular, `"\\"` shall match a
12516 backslash in *string*. If FNM_NOESCAPE is set, a backslash character shall be treated as an
12517 ordinary character.

12518 If FNM_PERIOD is set in *flags*, then a leading period (`'.'`) in *string* shall match a period in
12519 *pattern*; as described by rule 2 in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section
12520 2.13.3, Patterns Used for Filename Expansion where the location of ‘‘leading’’ is indicated by the
12521 value of FNM_PATHNAME:

12522 • If FNM_PATHNAME is set, a period is ‘‘leading’’ if it is the first character in *string* or if it
12523 immediately follows a slash.

12524 • If FNM_PATHNAME is not set, a period is ‘‘leading’’ only if it is the first character of *string*.

12525 If FNM_PERIOD is not set, then no special restrictions are placed on matching a period.

12526 **RETURN VALUE**

12527 If *string* matches the pattern specified by *pattern*, then *fnmatch*( ) shall return 0. If there is no
12528 match, *fnmatch*( ) shall return FNM_NOMATCH, which is defined in **<fnmatch.h>**. If an error
12529 occurs, *fnmatch*( ) shall return another non-zero value.

12530 **ERRORS**

12531 No errors are defined.

12532 **EXAMPLES**

12533 None.

12534 **APPLICATION USAGE**

12535 The *fnmatch*( ) function has two major uses. It could be used by an application or utility that
12536 needs to read a directory and apply a pattern against each entry. The *find* utility is an example of
12537 this. It can also be used by the *pax* utility to process its *pattern* operands, or by applications that
12538 need to match strings in a similar manner.

12539 The name *fnmatch*( ) is intended to imply *filename* match, rather than *pathname* match. The default
12540 action of this function is to match filenames, rather than pathnames, since it gives no special
12541 significance to the slash character. With the FNM_PATHNAME flag, *fnmatch*( ) does match
12542 pathnames, but without tilde expansion, parameter expansion, or special treatment for a period

at the beginning of a filename.

RATIONALE

This function replaced the REG_FILENAME flag of *regcomp*( ) in early proposals of this volume of IEEE Std 1003.1-200x. It provides virtually the same functionality as the *regcomp*( ) and *regexec*( ) functions using the REG_FILENAME and REG_FSLASH flags (the REG_FSLASH flag was proposed for *regcomp*( ), and would have had the opposite effect from FNM_PATHNAME), but with a simpler function and less system overhead.

FUTURE DIRECTIONS

None.

SEE ALSO

glob( ), *wordexp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fnmatch.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

Moved from POSIX2 C-language Binding to BASE.

12559 **NAME**

12560 fopen — open a stream

12561 **SYNOPSIS**

12562 `#include <stdio.h>`

12563 `FILE *fopen(const char *restrict filename, const char *restrict mode);`

12564 **DESCRIPTION**

12565 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12566 conflict between the requirements described here and the ISO C standard is unintentional. This
12567 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12568 The *fopen*( ) function shall open the file whose pathname is the string pointed to by *filename*, and |
12569 associates a stream with it.

12570 The *mode* argument points to a string. If the string is one of the following, the file shall be opened |
12571 in the indicated mode. Otherwise, the behavior is undefined. |

12572 *r* or *rb*	Open file for reading.
12573 *w* or *wb*	Truncate to zero length or create file for writing.
12574 *a* or *ab*	Append; open or create file for writing at end-of-file.
12575 *r+* or *rb+* or *r+b*	Open file for update (reading and writing).
12576 *w+* or *wb+* or *w+b*	Truncate to zero length or create file for update.
12577 *a+* or *ab+* or *a+b*	Append; open or create file for update, writing at end-of-file.

12578 CX The character `'b'` shall have no effect, but is allowed for ISO C standard conformance. Opening |
12579 a file with read mode (*r* as the first character in the *mode* argument) shall fail if the file does not
12580 exist or cannot be read.

12581 Opening a file with append mode (*a* as the first character in the *mode* argument) shall cause all
12582 subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening
12583 calls to *fseek*( ).

12584 When a file is opened with update mode (`'+'` as the second or third character in the *mode*
12585 argument), both input and output may be performed on the associated stream. However, the
12586 application shall ensure that output is not directly followed by input without an intervening call
12587 to *fflush*( ) or to a file positioning function (*fseek*( ), *fsetpos*( ), or *rewind*( )), and input is not directly
12588 followed by output without an intervening call to a file positioning function, unless the input
12589 operation encounters end-of-file.

12590 When opened, a stream is fully buffered if and only if it can be determined not to refer to an
12591 interactive device. The error and end-of-file indicators for the stream shall be cleared.

12592 CX If *mode* is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and the file did not previously exist, upon
12593 successful completion, *fopen*( ) function shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
12594 fields of the file and the *st_ctime* and *st_mtime* fields of the parent directory.

12595 If *mode* is *w*, *wb*, *w+*, *wb+*, or *w+b*, and the file did previously exist, upon successful completion,
12596 *fopen*( ) shall mark for update the *st_ctime* and *st_mtime* fields of the file. The *fopen*( ) function
12597 shall allocate a file descriptor as *open*( ) does.

12598 XSI After a successful call to the *fopen*( ) function, the orientation of the stream shall be cleared, the
12599 encoding rule shall be cleared, and the associated **mbstate_t** object shall be set to describe an
12600 initial conversion state.

12601 CX The largest value that can be represented correctly in an object of type **off_t** shall be established
12602 as the offset maximum in the open file description.

12603 **RETURN VALUE**

12604 Upon successful completion, *fopen*( ) shall return a pointer to the object controlling the stream.
12605 CX Otherwise, a null pointer shall be returned, and *errno* shall be set to indicate the error.

12606 **ERRORS**

12607 The *fopen*( ) function shall fail if:

12608 CX [EACCES] Search permission is denied on a component of the path prefix, or the file
12609 exists and the permissions specified by *mode* are denied, or the file does not
12610 exist and write permission is denied for the parent directory of the file to be
12611 created.

12612 CX [EINTR] A signal was caught during *fopen*( ).

12613 CX [EISDIR] The named file is a directory and *mode* requires write access.

12614 CX [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
12615 argument.

12616 CX [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

12617 CX [ENAMETOOLONG]
12618 The length of the *filename* argument exceeds {PATH_MAX} or a pathname |
12619 component is longer than {NAME_MAX}. |

12620 CX [ENFILE] The maximum allowable number of files is currently open in the system.

12621 CX [ENOENT] A component of *filename* does not name an existing file or *filename* is an empty
12622 string.

12623 CX [ENOSPC] The directory or file system that would contain the new file cannot be
12624 expanded, the file does not exist, and it was to be created.

12625 CX [ENOTDIR] A component of the path prefix is not a directory.

12626 CX [ENXIO] The named file is a character special or block special file, and the device
12627 associated with this special file does not exist.

12628 CX [EOVERFLOW] The named file is a regular file and the size of the file cannot be represented
12629 correctly in an object of type **off_t**.

12630 CX [EROFS] The named file resides on a read-only file system and *mode* requires write
12631 access.

12632 The *fopen*( ) function may fail if:

12633 CX [EINVAL] The value of the *mode* argument is not valid.

12634 CX [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
12635 resolution of the *path* argument.

12636 CX [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

12637 CX [EMFILE] {STREAM_MAX} streams are currently open in the calling process.

12638 CX [ENAMETOOLONG]
12639 Pathname resolution of a symbolic link produced an intermediate result |
12640 whose length exceeds {PATH_MAX}.

12641 CX [ENOMEM] Insufficient storage space is available.

12642 CX [ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *mode*
12643 requires write access.

12644 **EXAMPLES**

12645 **Opening a File**

12646 The following example tries to open the file named **file** for reading. The *fopen*( ) function returns
12647 a file pointer that is used in subsequent *fgets*( ) and *fclose*( ) calls. If the program cannot open the
12648 file, it just ignores it.

```
12649 #include <stdio.h>
12650 ...
12651 FILE *fp;
12652 ...
12653 void rgrep(const char *file)
12654 {
12655 ...
12656     if ((fp = fopen(file, "r")) == NULL)
12657         return;
12658 ...
12659 }
```

12660 **APPLICATION USAGE**

12661 None.

12662 **RATIONALE**

12663 None.

12664 **FUTURE DIRECTIONS**

12665 None.

12666 **SEE ALSO**

12667 *fclose*( ), *fdopen*( ), *freopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

12668 **CHANGE HISTORY**

12669 First released in Issue 1. Derived from Issue 1 of the SVID.

12670 **Issue 5**

12671 Large File Summit extensions are added.

12672 **Issue 6**

12673 Extensions beyond the ISO C standard are now marked.

12674 The following new requirements on POSIX implementations derive from alignment with the |
12675 Single UNIX Specification:

12676 • In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file
12677 description. This change is to support large files.

12678 • In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support
12679 large files.

12680 • The [ELOOP] mandatory error condition is added.

12681 • The [EINVAL], [EMFILE], [ENAMETOOLONG], [ENOMEM], and [ETXTBSY] optional error
12682 conditions are added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

- The prototype for *fopen*( ) is updated.
- The DESCRIPTION is updated to note that if the argument *mode* points to a string other than those listed, then the behavior is undefined.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

12690 **NAME**

12691 fork — create a new process

12692 **SYNOPSIS**

12693
```
#include <unistd.h>
```

12694
```
pid_t fork(void);
```

12695 **DESCRIPTION**

12696 The *fork*( ) function shall create a new process. The new process (child process) shall be an exact
12697 copy of the calling process (parent process) except as detailed below:

12698 • The child process shall have a unique process ID.

12699 • The child process ID also shall not match any active process group ID.

12700 • The child process shall have a different parent process ID, which shall be the process ID of
12701 the calling process.

12702 • The child process shall have its own copy of the parent's file descriptors. Each of the child's
12703 file descriptors shall refer to the same open file description with the corresponding file
12704 descriptor of the parent.

12705 • The child process shall have its own copy of the parent's open directory streams. Each open
12706 directory stream in the child process may share directory stream positioning with the
12707 corresponding directory stream of the parent.

12708 XSI • The child process shall have its own copy of the parent's message catalog descriptors.

12709 • The child process' values of *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* shall be set to 0.

12710 • The time left until an alarm clock signal shall be reset to zero, and the alarm, if any, shall be
12711 canceled; see *alarm*( ).

12712 XSI • All *semadj* values shall be cleared.

12713 • File locks set by the parent process shall not be inherited by the child process.

12714 • The set of signals pending for the child process shall be initialized to the empty set.

12715 XSI • Interval timers shall be reset in the child process.

12716 SEM • Any semaphores that are open in the parent process shall also be open in the child process.

12717 ML • The child process shall not inherit any address space memory locks established by the parent
12718 process via calls to *mlockall*( ) or *mlock*( ).

12719 MF|SHM • Memory mappings created in the parent shall be retained in the child process.
12720 MAP_PRIVATE mappings inherited from the parent shall also be MAP_PRIVATE mappings
12721 in the child, and any modifications to the data in these mappings made by the parent prior to
12722 calling *fork*( ) shall be visible to the child. Any modifications to the data in MAP_PRIVATE
12723 mappings made by the parent after *fork*( ) returns shall be visible only to the parent.
12724 Modifications to the data in MAP_PRIVATE mappings made by the child shall be visible only
12725 to the child.

12726 PS • For the SCHED_FIFO and SCHED_RR scheduling policies, the child process shall inherit the
12727 policy and priority settings of the parent process during a *fork*( ) function. For other
12728 scheduling policies, the policy and priority settings on *fork*( ) are implementation-defined.

12729 TMR • Per-process timers created by the parent shall not be inherited by the child process.

12730 MSG • The child process shall have its own copy of the message queue descriptors of the parent.
12731 Each of the message descriptors of the child shall refer to the same open message queue

12732 description as the corresponding message descriptor of the parent.

12733 AIO • No asynchronous input or asynchronous output operations shall be inherited by the child
12734 process.

12735 • A process shall be created with a single thread. If a multi-threaded process calls *fork*(), the
12736 new process shall contain a replica of the calling thread and its entire address space, possibly
12737 including the states of mutexes and other resources. Consequently, to avoid errors, the child
12738 process may only execute async-signal-safe operations until such time as one of the *exec*
12739 THR functions is called. Fork handlers may be established by means of the *pthread_atfork*()
12740 function in order to maintain application invariants across *fork*() calls.

12741 TRC TRI • If the Trace option and the Trace Inherit option are both supported:

12742 If the calling process was being traced in a trace stream that had its inheritance policy set to
12743 POSIX_TRACE_INHERITED, the child process shall be traced into that trace stream, and the
12744 child process shall inherit the parent's mapping of trace event names to trace event type
12745 identifiers. If the trace stream in which the calling process was being traced had its
12746 inheritance policy set to POSIX_TRACE_CLOSE_FOR_CHILD, the child process shall not be
12747 traced into that trace stream. The inheritance policy is set by a call to the
12748 *posix_trace_attr_setinherited*() function.

12749 TRC • If the Trace option is supported, but the Trace Inherit option is not supported:

12750 The child process shall not be traced into any of the trace streams of its parent process.

12751 TRC • If the Trace option is supported, the child process of a trace controller process shall not
12752 control the trace streams controlled by its parent process.

12753 CPT • The initial value of the CPU-time clock of the child process shall be set to zero.

12754 TCT • The initial value of the CPU-time clock of the single thread of the child process shall be set to
12755 zero.

12756 All other process characteristics defined by IEEE Std 1003.1-200x shall be the same in the parent
12757 and child processes. The inheritance of process characteristics not defined by
12758 IEEE Std 1003.1-200x is unspecified by IEEE Std 1003.1-200x.

12759 After *fork*(), both the parent and the child processes shall be capable of executing independently
12760 before either one terminates.

12761 **RETURN VALUE**

12762 Upon successful completion, *fork*() shall return 0 to the child process and shall return the
12763 process ID of the child process to the parent process. Both processes shall continue to execute
12764 from the *fork*() function. Otherwise, –1 shall be returned to the parent process, no child process
12765 shall be created, and *errno* shall be set to indicate the error.

12766 **ERRORS**

12767 The *fork*() function shall fail if:

12768 [EAGAIN] The system lacked the necessary resources to create another process, or the
12769 system-imposed limit on the total number of processes under execution
12770 system-wide or by a single user {CHILD_MAX} would be exceeded.

12771 The *fork*() function may fail if:

12772 [ENOMEM] Insufficient storage space is available.

12773 **EXAMPLES**

12774 None.

12775 **APPLICATION USAGE**

12776 None.

12777 **RATIONALE**

12778 Many historical implementations have timing windows where a signal sent to a process group
12779 (for example, an interactive SIGINT) just prior to or during execution of *fork*( ) is delivered to the
12780 parent following the *fork*( ) but not to the child because the *fork*( ) code clears the child's set of
12781 pending signals. This volume of IEEE Std 1003.1-200x does not require, or even permit, this
12782 behavior. However, it is pragmatic to expect that problems of this nature may continue to exist
12783 in implementations that appear to conform to this volume of IEEE Std 1003.1-200x and pass
12784 available verification suites. This behavior is only a consequence of the implementation failing to
12785 make the interval between signal generation and delivery totally invisible. From the
12786 application's perspective, a *fork*( ) call should appear atomic. A signal that is generated prior to
12787 the *fork*( ) should be delivered prior to the *fork*( ). A signal sent to the process group after the
12788 *fork*( ) should be delivered to both parent and child. The implementation may actually initialize
12789 internal data structures corresponding to the child's set of pending signals to include signals
12790 sent to the process group during the *fork*( ). Since the *fork*( ) call can be considered as atomic
12791 from the application's perspective, the set would be initialized as empty and such signals would
12792 have arrived after the *fork*( ); see also **<signal.h>**.

12793 One approach that has been suggested to address the problem of signal inheritance across *fork*( )
12794 is to add an [EINTR] error, which would be returned when a signal is detected during the call.
12795 While this is preferable to losing signals, it was not considered an optimal solution. Although it
12796 is not recommended for this purpose, such an error would be an allowable extension for an
12797 implementation.

12798 The [ENOMEM] error value is reserved for those implementations that detect and distinguish
12799 such a condition. This condition occurs when an implementation detects that there is not enough
12800 memory to create the process. This is intended to be returned when [EAGAIN] is inappropriate
12801 because there can never be enough memory (either primary or secondary storage) to perform the
12802 operation. Since *fork*( ) duplicates an existing process, this must be a condition where there is
12803 sufficient memory for one such process, but not for two. Many historical implementations
12804 actually return [ENOMEM] due to temporary lack of memory, a case that is not generally
12805 distinct from [EAGAIN] from the perspective of a conforming application.

12806 Part of the reason for including the optional error [ENOMEM] is because the SVID specifies it
12807 and it should be reserved for the error condition specified there. The condition is not applicable
12808 on many implementations.

12809 IEEE Std 1003.1-1988 neglected to require concurrent execution of the parent and child of *fork*( ).
12810 A system that single-threads processes was clearly not intended and is considered an
12811 unacceptable ''toy implementation'' of this volume of IEEE Std 1003.1-200x. The only objection
12812 anticipated to the phrase ''executing independently'' is testability, but this assertion should be
12813 testable. Such tests require that both the parent and child can block on a detectable action of the
12814 other, such as a write to a pipe or a signal. An interactive exchange of such actions should be
12815 possible for the system to conform to the intent of this volume of IEEE Std 1003.1-200x.

12816 The [EAGAIN] error exists to warn applications that such a condition might occur. Whether it
12817 occurs or not is not in any practical sense under the control of the application because the
12818 condition is usually a consequence of the user's use of the system, not of the application's code.
12819 Thus, no application can or should rely upon its occurrence under any circumstances, nor
12820 should the exact semantics of what concept of ''user'' is used be of concern to the application
12821 writer. Validation writers should be cognizant of this limitation.

There are two reasons why POSIX programmers call *fork*(). One reason is to create a new thread of control within the same program (which was originally only possible in POSIX by creating a new process); the other is to create a new process running a different program. In the latter case, the call to *fork*() is soon followed by a call to one of the *exec* functions.

The general problem with making *fork*() work in a multi-threaded world is what to do with all of the threads. There are two alternatives. One is to copy all of the threads into the new process. This causes the programmer or implementation to deal with threads that are suspended on system calls or that might be about to execute system calls that should not be executed in the new process. The other alternative is to copy only the thread that calls *fork*(). This creates the difficulty that the state of process-local resources is usually held in process memory. If a thread that is not calling *fork*() holds a resource, that resource is never released in the child process because the thread whose job it is to release the resource does not exist in the child process.

When a programmer is writing a multi-threaded program, the first described use of *fork*(), creating new threads in the same program, is provided by the *pthread_create*() function. The *fork*() function is thus used only to run new programs, and the effects of calling functions that require certain resources between the call to *fork*() and the call to an *exec* function are undefined.

The addition of the *forkall*() function to the standard was considered and rejected. The *forkall*() function lets all the threads in the parent be duplicated in the child. This essentially duplicates the state of the parent in the child. This allows threads in the child to continue processing and allows locks and the state to be preserved without explicit *pthread_atfork*() code. The calling process has to ensure that the threads processing state that is shared between the parent and child (that is, file descriptors or MAP_SHARED memory) behaves properly after *forkall*(). For example, if a thread is reading a file descriptor in the parent when *forkall*() is called, then two threads (one in the parent and one in the child) are reading the file descriptor after the *forkall*(). If this is not desired behavior, the parent process has to synchronize with such threads before calling *forkall*().

When *forkall*() is called, threads, other than the calling thread, that are in POSIX System Interfaces functions that can return with an [EINTR] error may have those functions return [EINTR] if the implementation cannot ensure that the function behaves correctly in the parent and child. In particular, *pthread_cond_wait*() and *pthread_cond_timedwait*() need to return in order to ensure that the condition has not changed. These functions can be awakened by a spurious condition wakeup rather than returning [EINTR].

FUTURE DIRECTIONS

None.

SEE ALSO

alarm(), *exec*, *fcntl*(), *posix_trace_attr_getinherited*(), *posix_trace_trid_eventid_open*(), *semop*(), *signal*(), *times*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is changed for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not

12869 required for UNIX applications.

12870 The following changes were made to align with the IEEE P1003.1a draft standard:

12871 • The effect of *fork*( ) on a pending alarm call in the child process is clarified.

12872 The description of CPU-time clock semantics is added for alignment with IEEE Std 1003.1d-1999.

12873 The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

12874 **NAME**

12875 fpathconf, pathconf — get configurable pathname variables

12876 **SYNOPSIS**

12877 `#include <unistd.h>`

12878 `long fpathconf(int fildes, int name);`
12879 `long pathconf(const char *path, int name);`

12880 **DESCRIPTION**

12881 The *fpathconf*( ) and *pathconf*( ) functions shall determine the current value of a configurable limit
12882 or option (*variable*) that is associated with a file or directory.

12883 For *pathconf*( ), the *path* argument points to the pathname of a file or directory.

12884 For *fpathconf*( ), the *fildes* argument is an open file descriptor.

12885 The *name* argument represents the variable to be queried relative to that file or directory.
12886 Implementations shall support all of the variables listed in the following table and may support
12887 others. The variables in the following table come from **<limits.h>** or **<unistd.h>** and the
12888 symbolic constants, defined in **<unistd.h>**, are the corresponding values used for *name*. Support
12889 for some pathname configuration variables is dependent on implementation options (see
12890 shading and margin codes in the table below). Where an implementation option is not
12891 supported, the variable need not be supported.

12892

12893

	Variable	Value of *name*	Requirements
12894	{FILESIZEBITS}	_PC_FILESIZEBITS	3, 4
12895	{LINK_MAX}	_PC_LINK_MAX	1
12896	{MAX_CANON}	_PC_MAX_CANON	2
12897	{MAX_INPUT}	_PC_MAX_INPUT	2
12898	{NAME_MAX}	_PC_NAME_MAX	3, 4
12899	{PATH_MAX}	_PC_PATH_MAX	4, 5
12900	{PIPE_BUF}	_PC_PIPE_BUF	6
12901 ADV	{POSIX_ALLOC_SIZE_MIN}	_PC_ALLOC_SIZE_MIN	
12902 ADV	{POSIX_REC_INCR_XFER_SIZE}	_PC_REC_INCR_XFER_SIZE	
12903 ADV	{POSIX_REC_MAX_XFER_SIZE}	_PC_REC_MAX_XFER_SIZE	
12904 ADV	{POSIX_REC_MIN_XFER_SIZE}	_PC_REC_MIN_XFER_SIZE	
12905 ADV	{POSIX_REC_XFER_ALIGN}	_PC_REC_XFER_ALIGN	
12906	{SYMLINK_MAX}	_PC_SYMLINK_MAX	4, 9
12907	_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	7
12908	_POSIX_NO_TRUNC	_PC_NO_TRUNC	3, 4
12909	_POSIX_VDISABLE	_PC_VDISABLE	2
12910	_POSIX_ASYNC_IO	_PC_ASYNC_IO	8
12911	_POSIX_PRIO_IO	_PC_PRIO_IO	8
12912	_POSIX_SYNC_IO	_PC_SYNC_IO	8

Requirements

1. If *path* or *fildes* refers to a directory, the value returned shall apply to the directory itself.
2. If *path* or *fildes* does not refer to a terminal file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
3. If *path* or *fildes* refers to a directory, the value returned shall apply to filenames within the directory.
4. If *path* or *fildes* does not refer to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
5. If *path* or *fildes* refers to a directory, the value returned shall be the maximum length of a relative pathname when the specified directory is the working directory.
6. If *path* refers to a FIFO, or *fildes* refers to a pipe or FIFO, the value returned shall apply to the referenced object. If *path* or *fildes* refers to a directory, the value returned shall apply to any FIFO that exists or can be created within the directory. If *path* or *fildes* refers to any other type of file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
7. If *path* or *fildes* refers to a directory, the value returned shall apply to any files, other than directories, that exist or can be created within the directory.
8. If *path* or *fildes* refers to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
9. If *path* or *fildes* refers to a directory, the value returned shall be the maximum length of the string that a symbolic link in that directory can contain.

RETURN VALUE

If *name* is an invalid value, both *pathconf*( ) and *fpathconf*( ) shall return –1 and set *errno* to indicate the error.

If the variable corresponding to *name* has no limit for the *path* or file descriptor, both *pathconf*( ) and *fpathconf*( ) shall return –1 without changing *errno*. If the implementation needs to use *path* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *path*, or if the process did not have appropriate privileges to query the file specified by *path*, or *path* does not exist, *pathconf*( ) shall return –1 and set *errno* to indicate the error.

If the implementation needs to use *fildes* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *fildes*, or if *fildes* is an invalid file descriptor, *fpathconf*( ) shall return –1 and set *errno* to indicate the error.

Otherwise, *pathconf*( ) or *fpathconf*( ) shall return the current variable value for the file or directory without changing *errno*. The value returned shall not be more restrictive than the corresponding value available to the application when it was compiled with the implementation's **<limits.h>** or **<unistd.h>**.

ERRORS

The *pathconf*( ) function shall fail if:

[EINVAL] The value of *name* is not valid.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

The *pathconf*( ) function may fail if:

[EACCES] Search permission is denied for a component of the path prefix.

[EINVAL] The implementation does not support an association of the variable *name* with the specified file.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

The *fpathconf*( ) function shall fail if:

[EINVAL] The value of *name* is not valid.

The *fpathconf*( ) function may fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] The implementation does not support an association of the variable *name* with the specified file.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *pathconf*( ) function was proposed immediately after the *sysconf*( ) function when it was realized that some configurable values may differ across file system, directory, or device boundaries.

For example, {NAME_MAX} frequently changes between System V and BSD-based file systems; System V uses a maximum of 14, BSD 255. On an implementation that provides both types of file systems, an application would be forced to limit all pathname components to 14 bytes, as this would be the value specified in **<limits.h>** on such a system.

Therefore, various useful values can be queried on any pathname or file descriptor, assuming that the appropriate permissions are in place.

The value returned for the variable {PATH_MAX} indicates the longest relative pathname that could be given if the specified directory is the process' current working directory. A process may not always be able to generate a name that long and use it if a subdirectory in the pathname crosses into a more restrictive file system.

The value returned for the variable _POSIX_CHOWN_RESTRICTED also applies to directories that do not have file systems mounted on them. The value may change when crossing a mount point, so applications that need to know should check for each directory. (An even easier check is to try the *chown*( ) function and look for an error in case it happens.)

Unlike the values returned by *sysconf*( ), the pathname-oriented variables are potentially more volatile and are not guaranteed to remain constant throughout the process' lifetime. For

example, in between two calls to *pathconf*( ), the file system in question may have been unmounted and remounted with different characteristics.

Also note that most of the errors are optional. If one of the variables always has the same value on an implementation, the implementation need not look at *path* or *fildes* to return that value and is, therefore, not required to detect any of the errors except the meaning of [EINVAL] that indicates that the value of *name* is not valid for that variable.

If the value of any of the limits are unspecified (logically infinite), they will not be defined in **<limits.h>** and the *pathconf*( ) and *fpathconf*( ) functions return –1 without changing *errno*. This can be distinguished from the case of giving an unrecognized *name* argument because *errno* is set to [EINVAL] in this case.

Since –1 is a valid return value for the *pathconf*( ) and *fpathconf*( ) functions, applications should set *errno* to zero before calling them and check *errno* only if the return value is –1.

For the case of {SYMLINK_MAX}, since both *pathconf*( ) and *open*( ) follow symbolic links, there is no way that *path* or *fildes* could refer to a symbolic link.

FUTURE DIRECTIONS

None.

SEE ALSO

confstr( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<unistd.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to include {FILESIZEBITS}.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The _PC_SYMLINK_MAX entry is added to the table in the DESCRIPTION.

The *pathconf*( ) variables {POSIX_ALLOC_SIZE_MIN}, {POSIX_REC_INCR_XFER_SIZE}, {POSIX_REC_MAX_XFER_SIZE}, {POSIX_REC_MIN_XFER_SIZE}, {POSIX_REC_XFER_ALIGN} and their associated names are added for alignment with IEEE Std 1003.1d-1999.

13036 **NAME**

13037 fpclassify — classify real floating type

13038 **SYNOPSIS**

13039 `#include <math.h>`

13040 `int fpclassify(real-floating x);`

13041 **DESCRIPTION**

13042 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13043 conflict between the requirements described here and the ISO C standard is unintentional. This
13044 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13045 The *fpclassify*( ) macro shall classify its argument value as NaN, infinite, normal, subnormal,
13046 zero, or into another implementation-defined category. First, an argument represented in a
13047 format wider than its semantic type is converted to its semantic type. Then classification is based
13048 on the type of the argument.

13049 **RETURN VALUE**

13050 The *fpclassify*( ) macro shall return the value of the number classification macro appropriate to
13051 the value of its argument.

13052 **ERRORS**

13053 No errors are defined.

13054 **EXAMPLES**

13055 None.

13056 **APPLICATION USAGE**

13057 None.

13058 **RATIONALE**

13059 None.

13060 **FUTURE DIRECTIONS**

13061 None.

13062 **SEE ALSO**

13063 *isfinite*( ), *isinf*( ), *isnan*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of
13064 IEEE Std 1003.1-200x, **<math.h>**

13065 **CHANGE HISTORY**

13066 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

13067 **NAME**

13068 fprintf, printf, snprintf, sprintf — print formatted output

13069 **SYNOPSIS**

13070
```
#include <stdio.h>

int fprintf(FILE *restrict stream, const char *restrict format, ...);
int printf(const char *restrict format, ...);
int snprintf(char *restrict s, size_t n,
    const char *restrict format, ...);
int sprintf(char *restrict s, const char *restrict format, ...);
```

13076 **DESCRIPTION**

13077 CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13080 The *fprintf*( ) function shall place output on the named output *stream*. The *printf*( ) function shall place output on the standard output stream *stdout*. The *sprintf*( ) function shall place output followed by the null byte, `'\0'`, in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough space is available.

13084 The *snprintf*( ) function shall be equivalent to *sprintf*( ), with the addition of the *n* argument which states the size of the buffer referred to by *s*. If *n* is zero, nothing shall be written and *s* may be a null pointer. Otherwise, output bytes beyond the *n*-1st shall be discarded instead of being written to the array, and a null byte is written at the end of the bytes actually written into the array.

13089 If copying takes place between objects that overlap as a result of a call to *sprintf*( ) or *snprintf*( ), the results are undefined.

13091 Each of these functions converts, formats, and prints its arguments under control of the *format*. The *format* is a character string, beginning and ending in its initial shift state, if any. The *format* is composed of zero or more directives: *ordinary characters*, which are simply copied to the output stream, and *conversion specifications*, each of which shall result in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the excess arguments shall be evaluated but are otherwise ignored.

13098 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion specifier character % (see below) is replaced by the sequence "%*n*$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

13104 The *format* can contain either numbered argument conversion specifications (that is, "%*n*$" and "**m*$"), or unnumbered argument conversion specifications (that is, % and *), but not both. The only exception to this is that %% can be mixed with the "%*n*$" form. The results of mixing numbered and unnumbered argument specifications in a *format* string are undefined. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*–*1*)th, are specified in the format string.

13110 In format strings containing the "%*n*$" form of conversion specification, numbered arguments in the argument list can be referenced from the format string as many times as required.

13112 In format strings containing the % form of conversion specification, each argument in the argument list is used exactly once.

13114 CX All forms of the *fprintf*() functions allow for the insertion of a language-dependent radix
13115 character in the output string. The radix character is defined in the program's locale (category
13116 *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the
13117 radix character shall default to a period (`'.'`).

13118 XSI Each conversion specification is introduced by the `'%'` character or by the character sequence
13119 `"%`*n*`$"`, after which the following appear in sequence:

13120 - Zero or more *flags* (in any order), which modify the meaning of the conversion specification.

13121 - An optional minimum *field width*. If the converted value has fewer bytes than the field
13122 width, it shall be padded with spaces by default on the left; it shall be padded on the right if |
13123 the left-adjustment flag (`'-'`), described below, is given to the field width. The field width |
13124 takes the form of an asterisk (`'*'`), described below, or a decimal integer.

13125 - An optional *precision* that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`,
13126 and `X` conversion specifiers; the number of digits to appear after the radix character for the `a`,
13127 `A`, `e`, `E`, `f`, and `F` conversion specifiers; the maximum number of significant digits for the `g`
13128 XSI and `G` conversion specifiers; or the maximum number of bytes to be printed from a string in `s`
13129 and `S` conversion specifiers. The precision takes the form of a period (`'.'`) followed either by
13130 an asterisk (`'*'`), described below, or an optional decimal digit string, where a null digit
13131 string is treated as zero. If a precision appears with any other conversion specifier, the
13132 behavior is undefined.

13133 - An optional length modifier that specifies the size of the argument.

13134 - A *conversion specifier* character that indicates the type of conversion to be applied.

13135 A field width, or precision, or both, may be indicated by an asterisk (`'*'`). In this case an
13136 argument of type **int** supplies the field width or precision. Applications shall ensure that
13137 arguments specifying field width, or precision, or both appear in that order before the argument,
13138 if any, to be converted. A negative field width is taken as a `'-'` flag followed by a positive field
13139 XSI width. A negative precision is taken as if the precision were omitted. In format strings
13140 containing the `"%`*n*`$"` form of a conversion specification, a field width or precision may be
13141 indicated by the sequence `"*`*m*`$"`, where *m* is a decimal integer in the range [1,{NL_ARGMAX}]
13142 giving the position in the argument list (after the format argument) of an integer argument
13143 containing the field width or precision, for example:

13144 ```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

13145 The flag characters and their meanings are:

13146 XSI `'` The integer portion of the result of a decimal conversion (`%i`, `%d`, `%u`, `%f`, `%F`, `%g`, or `%G`)
13147 shall be formatted with thousands' grouping characters. For other conversions the
13148 behavior is undefined. The non-monetary grouping character is used.

13149 `-` The result of the conversion shall be left-justified within the field. The conversion is
13150 right-justified if this flag is not specified.

13151 `+` The result of a signed conversion shall always begin with a sign (`'+'` or `'-'`). The
13152 conversion shall begin with a sign only when a negative value is converted if this flag is
13153 not specified.

13154 <space> If the first character of a signed conversion is not a sign or if a signed conversion results
13155 in no characters, a <space> shall be prefixed to the result. This means that if the
13156 <space> and `'+'` flags both appear, the <space> flag shall be ignored.

13157 `#` Specifies that the value is to be converted to an alternative form. For `o` conversion, it |
13158 increases the precision (if necessary) to force the first digit of the result to be zero. For `x`
```

or X conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For a, A, e, E, f, F, g, and G conversion specifiers, the result shall always contain a radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversion specifiers, trailing zeros shall *not* be removed from the result as they normally are. For other conversion specifiers, the behavior is undefined.

`0` For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversion specifiers, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the '0' and '−' flags both appear, the '0' flag is ignored. For d, i, o, u, x, and X conversion specifiers, if a precision is specified, the '0' flag is ignored. XSI If the '0' and '\'' flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined.

The length modifiers and their meanings are:

`hh` Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **signed char** or **unsigned char** before printing); or that a following n conversion specifier applies to a pointer to a **signed char** argument.

`h` Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **short** or **unsigned short** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **short** or **unsigned short** before printing); or that a following n conversion specifier applies to a pointer to a **short** argument.

`l` (ell) Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **long** or **unsigned long** argument; that a following n conversion specifier applies to a pointer to a **long** argument; that a following c conversion specifier applies to a **wint_t** argument; that a following s conversion specifier applies to a pointer to a **wchar_t** argument; or has no effect on a following a, A, e, E, f, F, g, or G conversion specifier.

`ll` (ell-ell) Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **long long** or **unsigned long long** argument; or that a following n conversion specifier applies to a pointer to a **long long** argument.

`j` Specifies that a following d, i, o, u, x, or X conversion specifier applies to an **intmax_t** or **uintmax_t** argument; or that a following n conversion specifier applies to a pointer to an **intmax_t** argument.

`z` Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **size_t** or the corresponding signed integer type argument; or that a following n conversion specifier applies to a pointer to a signed integer type corresponding to **size_t** argument.

`t` Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **ptrdiff_t** or the corresponding **unsigned** type argument; or that a following n conversion specifier applies to a pointer to a **ptrdiff_t** argument.

`L` Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a **long double** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

`d`, `i` The **int** argument shall be converted to a signed decimal in the style "[−]*dddd*". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

`o` The **unsigned** argument shall be converted to unsigned octal format in the style "*dddd*". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

`u` The **unsigned** argument shall be converted to unsigned decimal format in the style "*dddd*". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

`x` The **unsigned** argument shall be converted to unsigned hexadecimal format in the style "*dddd*"; the letters "abcdef" are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

`X` Equivalent to the `x` conversion specifier, except that letters "ABCDEF" are used instead of "abcdef".

`f`, `F` The **double** argument shall be converted to decimal notation in the style "[−]*ddd.ddd*", where the number of digits after the radix character is equal to the precision specification. If the precision is missing, it shall be taken as 6; if the precision is explicitly zero and no '#' flag is present, no radix character shall appear. If a radix character appears, at least one digit appears before it. The low-order digit shall be rounded in an implementation-defined manner.

A **double** argument representing an infinity shall be converted in one of the styles "[−]inf" or "[−]infinity"; which style is implementation-defined. A **double** argument representing a NaN shall be converted in one of the styles "[−]nan(*n-char-sequence*)"; or "[−]nan" which style, and the meaning of any *n-char-sequence*, is implementation-defined. The `F` conversion specifier produces "INF", "INFINITY", or "NAN" instead of "inf", "infinity", or "nan", respectively.

`e`, `E` The **double** argument shall be converted in the style "[−]*d.ddd*e±*dd*", where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it shall be taken as 6; if the precision is zero and no '#' flag is present, no radix character shall appear. The low-order digit shall be rounded in an implementation-defined manner. The `E` conversion specifier shall produce a number with 'E' instead of 'e' introducing the exponent. The exponent shall always contain at least two digits. If the value is zero, the exponent shall be zero.

A **double** argument representing an infinity or NaN shall be converted in the style of an `f` or `F` conversion specifier.

`g`, `G` The **double** argument shall be converted in the style `f` or `e` (or in the style `E` in the case of a `G` conversion specifier), with the precision specifying the number of significant

digits. If an explicit precision is zero, it shall be taken as 1. The style used depends on the value converted; style e (or E) shall be used only if the exponent resulting from such a conversion is less than –4 or greater than or equal to the precision. Trailing zeros shall be removed from the fractional portion of the result; a radix character shall appear only if it is followed by a digit.

A **double** argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

a, A — A **double** argument representing a floating-point number shall be converted in the style "[–]0x*h*.*hhhh*p±*d*", where there is one hexadecimal digit (which shall be non-zero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and FLT_RADIX is a power of 2, then the precision shall be sufficient for an exact representation of the value; if the precision is missing and FLT_RADIX is not a power of 2, then the precision shall be sufficient to distinguish values of type **double**, except that trailing zeros may be omitted; if the precision is zero and the '#' flag is not specified, no decimal-point character shall appear. The letters "abcdef" shall be used for a conversion and the letters "ABCDEF" for A conversion. The A conversion specifier produces a number with 'X' and 'P' instead of 'x' and 'p'. The exponent shall always contain at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent shall be zero.

A **double** argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

c — The **int** argument shall be converted to an **unsigned char**, and the resulting byte shall be written.

If an l (ell) qualifier is present, the **wint_t** argument shall be converted as if by an ls conversion specification with no precision and an argument that points to a two-element array of type **wchar_t**, the first element of which contains the **wint_t** argument to the ls conversion specification and the second element contains a null wide character.

s — The argument shall be a pointer to an array of **char**. Bytes from the array shall be written up to (but not including) any terminating null byte. If the precision is specified, no more than that many bytes shall be written. If the precision is not specified or is greater than the size of the array, the application shall ensure that the array contains a null byte.

If an l (ell) qualifier is present, the argument shall be a pointer to an array of type **wchar_t**. Wide characters from the array shall be converted to characters (each as if by a call to the *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting characters shall be written up to (but not including) the terminating null character (byte). If no precision is specified, the application shall ensure that the array contains a null wide character. If a precision is specified, no more than that many characters (bytes) shall be written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case shall a partial character written.

13298 p The argument shall be a pointer to **void**. The value of the pointer is converted to a
13299 sequence of printable characters, in an implementation-defined manner.

13300 n The argument shall be a pointer to an integer into which is written the number of bytes
13301 written to the output so far by this call to one of the *fprintf*( ) functions. No argument is
13302 converted.

13303 XSI C Equivalent to lc. |

13304 XSI S Equivalent to ls. |

13305 % Print a '%' character; no argument is converted. The complete conversion specification |
13306 shall be %%.

13307 If a conversion specification does not match one of the above forms, the behavior is undefined. If
13308 any argument is not the correct type for the corresponding conversion specification, the
13309 behavior is undefined.

13310 In no case shall a nonexistent or small field width cause truncation of a field; if the result of a |
13311 conversion is wider than the field width, the field shall be expanded to contain the conversion |
13312 result. Characters generated by *fprintf*( ) and *printf*( ) are printed as if *fputc*( ) had been called. |

13313 For the a and A conversion specifiers, if FLT_RADIX is a power of 2, the value shall be correctly |
13314 rounded to a hexadecimal floating number with the given precision.

13315 For a and A conversions, if FLT_RADIX is not a power of 2 and the result is not exactly |
13316 representable in the given precision, the result should be one of the two adjacent numbers in |
13317 hexadecimal floating style with the given precision, with the extra stipulation that the error |
13318 should have a correct sign for the current rounding direction. |

13319 For the e, E, f, F, g, and G conversion specifiers, if the number of significant decimal digits is at
13320 most DECIMAL_DIG, then the result should be correctly rounded. If the number of significant
13321 decimal digits is more than DECIMAL_DIG but the source value is exactly representable with
13322 DECIMAL_DIG digits, then the result should be an exact representation with trailing zeros.
13323 Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having
13324 DECIMAL_DIG significant digits; the value of the resultant decimal string D should satisfy $L <= D <= U$, with the extra stipulation that the error should have a correct sign for the current
13325
13326 rounding direction.

13327 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the call to a
13328 successful execution of *fprintf*( ) or *printf*( ) and the next successful completion of a call to *fflush*( )
13329 or *fclose*( ) on the same stream or a call to *exit*( ) or *abort*( ).

13330 **RETURN VALUE**

13331 Upon successful completion, the *fprintf*( ) and *printf*( ) functions shall return the number of bytes
13332 transmitted.

13333 Upon successful completion, the *sprintf*( ) function shall return the number of bytes written to *s*,
13334 excluding the terminating null byte.

13335 Upon successful completion, the *snprintf*( ) function shall return the number of bytes that would
13336 be written to *s* had *n* been sufficiently large excluding the terminating null byte.

13337 If an output error was encountered, these functions shall return a negative value.

13338 If the value of *n* is zero on a call to *snprintf*( ), nothing shall be written, the number of bytes that
13339 would have been written had *n* been sufficiently large excluding the terminating null shall be
13340 returned, and *s* may be a null pointer.

13341 **ERRORS**

13342 For the conditions under which *fprintf*() and *printf*() fail and may fail, refer to *fputc*() or
13343 *fputwc*().

13344 In addition, all forms of *fprintf*() may fail if:

13345 XSI [EILSEQ] A wide-character code that does not correspond to a valid character has been
13346 detected.

13347 XSI [EINVAL] There are insufficient arguments.

13348 The *printf*() and *fprintf*() functions may fail if:

13349 XSI [ENOMEM] Insufficient storage space is available.

13350 The *snprintf*() function shall fail if:

13351 XSI [EOVERFLOW] The value of *n* is greater than {INT_MAX} or the number of bytes needed to
13352 hold the output excluding the terminating null is greater than {INT_MAX}.

13353 **EXAMPLES**

13354 **Printing Language-Independent Date and Time**

13355 The following statement can be used to print date and time using a language-independent
13356 format:

13357 ```
printf(format, weekday, month, day, hour, min);
```

13358 For American usage, *format* could be a pointer to the following string:

13359 ```
"%s, %s %d, %d:%.2d\n"
```

13360 This example would produce the following message:

13361 ```
Sunday, July 3, 10:02
```

13362 For German usage, *format* could be a pointer to the following string:

13363 ```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

13364 This definition of *format* would produce the following message:

13365 ```
Sonntag, 3. Juli, 10:02
```

13366 **Printing File Information**

13367 The following example prints information about the type, permissions, and number of links of a
13368 specific file in a directory.

13369 The first two calls to *printf*() use data decoded from a previous *stat*() call. The user-defined
13370 *strperm*() function shall return a string similar to the one at the beginning of the output for the
13371 following command:

13372 ```
ls –l
```

13373 The next call to *printf*() outputs the owner's name if it is found using *getpwuid*(); the *getpwuid*()
13374 function shall return a **passwd** structure from which the name of the user is extracted. If the user
13375 name is not found, the program instead prints out the numeric value of the user ID.

13376 The next call prints out the group name if it is found using *getgrgid*(); *getgrgid*() is very similar to
13377 *getpwuid*() except that it shall return group information based on the group number. Once
13378 again, if the group is not found, the program prints the numeric value of the group for the entry.

The final call to *printf*( ) prints the size of the file.

```
#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>

char *strperm (mode_t);
...
struct stat statbuf;
struct passwd *pwd;
struct group *grp;
...
printf("%10.10s", strperm (statbuf.st_mode));
printf("%4d", statbuf.st_nlink);

if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
    printf(" %-8.8s", pwd->pw_name);
else
    printf(" %-8ld", (long) statbuf.st_uid);

if ((grp = getgrgid(statbuf.st_gid)) != NULL)
    printf(" %-8.8s", grp->gr_name);
else
    printf(" %-8ld", (long) statbuf.st_gid);

printf("%9jd", (intmax_t) statbuf.st_size);
...
```

Printing a Localized Date String

The following example gets a localized date string. The *nl_langinfo*( ) function shall return the localized date string, which specifies the order and layout of the date. The *strftime*( ) function takes this information and, using the **tm** structure for values, places the date and time information into *datestring*. The *printf*( ) function then outputs *datestring* and the name of the entry.

```
#include <stdio.h>
#include <time.h>
#include <langinfo.h>
...
struct dirent *dp;
struct tm *tm;
char datestring[256];
...
strftime(datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);

printf(" %s %s\n", datestring, dp->d_name);
...
```

13419 **Printing Error Information**

13420 The following example uses *fprintf*( ) to write error information to standard error.

13421 In the first group of calls, the program tries to open the password lock file named **LOCKFILE**. If
13422 the file already exists, this is an error, as indicated by the O_EXCL flag on the *open*( ) function. If
13423 the call fails, the program assumes that someone else is updating the password file, and the
13424 program exits.

13425 The next group of calls saves a new password file as the current password file by creating a link
13426 between **LOCKFILE** and the new password file **PASSWDFILE**.

```
13427 #include <sys/types.h>
13428 #include <sys/stat.h>
13429 #include <fcntl.h>
13430 #include <stdio.h>
13431 #include <stdlib.h>
13432 #include <unistd.h>
13433 #include <string.h>
13434 #include <errno.h>

13435 #define LOCKFILE "/etc/ptmp"
13436 #define PASSWDFILE "/etc/passwd"
13437 ...
13438 int pfd;
13439 ...
13440 if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
13441     S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == –1)
13442 {
13443     fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
13444     exit(1);
13445 }
13446 ...
13447 if (link(LOCKFILE,PASSWDFILE) == -1) {
13448     fprintf(stderr, "Link error: %s\n", strerror(errno));
13449     exit(1);
13450 }
13451 ...
```

13452 **Printing Usage Information**

13453 The following example checks to make sure the program has the necessary arguments, and uses
13454 *fprintf*( ) to print usage information if the expected number of arguments is not present.

```
13455 #include <stdio.h>
13456 #include <stdlib.h>
13457 ...
13458 char *Options = "hdbtl";
13459 ...
13460 if (argc < 2) {
13461     fprintf(stderr, "Usage: %s -%s <file\n", argv[0], Options); exit(1);
13462 }
13463 ...
```

13464 **Formatting a Decimal String**

13465 The following example prints a key and data pair on *stdout*. Note use of the '*' (asterisk) in the
13466 format string; this ensures the correct number of decimal places for the element based on the
13467 number of elements requested.

```
13468 #include <stdio.h>
13469 ...
13470 long i;
13471 char *keystr;
13472 int elementlen, len;
13473 ...
13474 while (len < elementlen) {
13475 ...
13476     printf("%s Element%0*ld\n", keystr, elementlen, i);
13477 ...
13478 }
```

13479 **Creating a Filename**

13480 The following example creates a filename using information from a previous *getpwnam*( )
13481 function that returned the HOME directory of the user.

```
13482 #include <stdio.h>
13483 #include <sys/types.h>
13484 #include <unistd.h>
13485 ...
13486 char filename[PATH_MAX+1];
13487 struct passwd *pw;
13488 ...
13489 sprintf(filename, "%s/%d.out", pw->pw_dir, getpid());
13490 ...
```

13491 **Reporting an Event**

13492 The following example loops until an event has timed out. The *pause*( ) function waits forever
13493 unless it receives a signal. The *fprintf*( ) statement should never occur due to the possible return
13494 values of *pause*( ).

```
13495 #include <stdio.h>
13496 #include <unistd.h>
13497 #include <string.h>
13498 #include <errno.h>
13499 ...
13500 while (!event_complete) {
13501 ...
13502     if (pause() != –1 || errno != EINTR)
13503         fprintf(stderr, "pause: unknown error: %s\n", strerror(errno));
13504 }
13505 ...
```

Printing Monetary Information

The following example uses *strfmon*( ) to convert a number and store it as a formatted monetary string named *convbuf*. If the first number is printed, the program prints the format and the description; otherwise, it just prints the number.

```
#include <monetary.h>
#include <stdio.h>
...
struct tblfmt {
    char *format;
    char *description;
};

struct tblfmt table[] = {
    { "%n", "default formatting" },
    { "%11n", "right align within an 11 character field" },
    { "%#5n", "aligned columns for values up to 99,999" },
    { "%=*#5n", "specify a fill character" },
    { "%=0#5n", "fill characters do not use grouping" },
    { "%^#5n", "disable the grouping separator" },
    { "%^#5.0n", "round off to whole units" },
    { "%^#5.4n", "increase the precision" },
    { "%(#5n", "use an alternative pos/neg style" },
    { "%!(#5n", "disable the currency symbol" },
};
...
float input[3];
int i, j;
char convbuf[100];
...
strfmon(convbuf, sizeof(convbuf), table[i].format, input[j]);

if (j == 0) {
    printf("%s%s%s\n", table[i].format,
        convbuf, table[i].description);
}
else {
    printf("%s\n", convbuf);
}
...
```

APPLICATION USAGE

If the application calling *fprintf*( ) has any objects of type **wint_t** or **wchar_t**, it must also include the **<wchar.h>** header to have these objects defined.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fputc( ), *fscanf*( ), *setlocale*( ), *wcrtomb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

13553 **CHANGE HISTORY**

13554 First released in Issue 1. Derived from Issue 1 of the SVID.

13555 **Issue 5**

13556 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the `l` (ell) qualifier can
13557 now be used with `c` and `s` conversion specifiers.

13558 The *snprintf*( ) function is new in Issue 5.

13559 **Issue 6**

13560 Extensions beyond the ISO C standard are now marked.

13561 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

13562 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

13563 • The prototypes for *fprintf*( ), *printf*( ), *snprintf*( ), and *sprintf*( ) are updated, and the XSI
13564 shading is removed from *snprintf*( ).

13565 • The description of *snprintf*( ) is aligned with the ISO C standard. Note that this supersedes |
13566 the *snprintf*( ) description in The Open Group Base Resolution bwg98-006, which changed the |
13567 behavior from Issue 5. |

13568 • The DESCRIPTION is updated. |

13569 The DESCRIPTION is updated to use the terms ‘‘conversion specifier’’ and ‘‘conversion
13570 specification’’ consistently. |

13571 ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated. |

13572 **NAME**

13573 fputc — put a byte on a stream

13574 **SYNOPSIS**

13575 `#include <stdio.h>`

13576 `int fputc(int c, FILE *stream);`

13577 **DESCRIPTION**

13578 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13579 conflict between the requirements described here and the ISO C standard is unintentional. This
13580 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13581 The *fputc*( ) function shall write the byte specified by *c* (converted to an **unsigned char**) to the
13582 output stream pointed to by *stream*, at the position indicated by the associated file-position
13583 indicator for the stream (if defined), and shall advance the indicator appropriately. If the file |
13584 cannot support positioning requests, or if the stream was opened with append mode, the byte |
13585 shall be appended to the output stream. |

13586 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
13587 execution of *fputc*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same
13588 stream or a call to *exit*( ) or *abort*( ).

13589 **RETURN VALUE**

13590 Upon successful completion, *fputc*( ) shall return the value it has written. Otherwise, it shall
13591 CX return EOF, the error indicator for the stream shall be set, and *errno* shall be set to indicate the
13592 error.

13593 **ERRORS**

13594 The *fputc*( ) function shall fail if either the *stream* is unbuffered or the *stream*'s buffer needs to be
13595 flushed, and:

13596 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
13597 process would be delayed in the write operation.

13598 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
13599 writing.

13600 CX [EFBIG] An attempt was made to write to a file that exceeds the maximum file size.

13601 XSI [EFBIG] An attempt was made to write to a file that exceeds the process' file size limit.

13602 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
13603 offset maximum.

13604 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data
13605 was transferred.

13606 CX [EIO] A physical I/O error has occurred, or the process is a member of a
13607 background process group attempting to write to its controlling terminal,
13608 TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the
13609 process group of the process is orphaned. This error may also be returned
13610 under implementation-defined conditions.

13611 CX [ENOSPC] There was no free space remaining on the device containing the file.

13612 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
13613 any process. A SIGPIPE signal shall also be sent to the thread.

13614 The *fputc*( ) function may fail if:

13615 CX [ENOMEM] Insufficient storage space is available.

13616 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
13617 capabilities of the device.

13618 **EXAMPLES**

13619 None.

13620 **APPLICATION USAGE**

13621 None.

13622 **RATIONALE**

13623 None.

13624 **FUTURE DIRECTIONS**

13625 None.

13626 **SEE ALSO**

13627 *ferror*( ), *fopen*( ), *getrlimit*( ), *putc*( ), *puts*( ), *setbuf*( ), *ulimit*( ), the Base Definitions volume of
13628 IEEE Std 1003.1-200x, **<stdio.h>**

13629 **CHANGE HISTORY**

13630 First released in Issue 1. Derived from Issue 1 of the SVID.

13631 **Issue 5**

13632 Large File Summit extensions are added.

13633 **Issue 6**

13634 Extensions beyond the ISO C standard are now marked.

13635 The following new requirements on POSIX implementations derive from alignment with the
13636 Single UNIX Specification:

13637 - The [EIO] and [EFBIG] mandatory error conditions are added.

13638 - The [ENOMEM] and [ENXIO] optional error conditions are added.

13639 **NAME**

13640 fputs — put a string on a stream

13641 **SYNOPSIS**

13642 `#include <stdio.h>`

13643 `int fputs(const char *restrict s, FILE *restrict stream);`

13644 **DESCRIPTION**

13645 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13646 conflict between the requirements described here and the ISO C standard is unintentional. This
13647 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13648 The *fputs*( ) function shall write the null-terminated string pointed to by *s* to the stream pointed
13649 to by *stream*. The terminating null byte shall not be written.

13650 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
13651 execution of *fputs*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same
13652 stream or a call to *exit*( ) or *abort*( ).

13653 **RETURN VALUE**

13654 Upon successful completion, *fputs*( ) shall return a non-negative number. Otherwise, it shall
13655 CX return EOF, set an error indicator for the stream, and set *errno* to indicate the error.

13656 **ERRORS**

13657 Refer to *fputc*( ).

13658 **EXAMPLES**

13659 **Printing to Standard Output**

13660 The following example gets the current time, converts it to a string using *localtime*( ) and
13661 *asctime*( ), and prints it to standard output using *fputs*( ). It then prints the number of minutes to
13662 an event for which it is waiting.

```
13663 #include <time.h>
13664 #include <stdio.h>
13665 ...
13666 time_t now;
13667 int minutes_to_event;
13668 ...
13669 time(&now);
13670 printf("The time is ");
13671 fputs(asctime(localtime(&now)), stdout);
13672 printf("There are still %d minutes to the event.\n",
13673     minutes_to_event);
13674 ...
```

13675 **APPLICATION USAGE**

13676 The *puts*( ) function appends a <newline> while *fputs*( ) does not.

13677 **RATIONALE**

13678 None.

13679 **FUTURE DIRECTIONS**

13680 None.

13681 **SEE ALSO**

13682 *fopen*( ), *putc*( ), *puts*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

13683 **CHANGE HISTORY**

13684 First released in Issue 1. Derived from Issue 1 of the SVID.

13685 **Issue 6**

13686 Extensions beyond the ISO C standard are now marked.

13687 The *fputs*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

13688 **NAME**

13689 fputwc — put a wide-character code on a stream

13690 **SYNOPSIS**

13691 ```
#include <stdio.h>
```

13692 ```
#include <wchar.h>
```

13693 ```
wint_t fputwc(wchar_t wc, FILE *stream);
```

13694 **DESCRIPTION**

13695 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13696 conflict between the requirements described here and the ISO C standard is unintentional. This
13697 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13698 The *fputwc*( ) function shall write the character corresponding to the wide-character code *wc* to
13699 the output stream pointed to by *stream*, at the position indicated by the associated file-position
13700 indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot
13701 support positioning requests, or if the stream was opened with append mode, the character is
13702 appended to the output stream. If an error occurs while writing the character, the shift state of
13703 the output file is left in an undefined state.

13704 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
13705 execution of *fputwc*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the
13706 same stream or a call to *exit*( ) or *abort*( ).

13707 **RETURN VALUE**

13708 Upon successful completion, *fputwc*( ) shall return *wc*. Otherwise, it shall return WEOF, the error
13709 CX indicator for the stream shall be set set, and *errno* shall be set to indicate the error.

13710 **ERRORS**

13711 The *fputwc*( ) function shall fail if either the stream is unbuffered or data in the *stream*'s buffer
13712 needs to be written, and:

13713 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
13714 process would be delayed in the write operation.

13715 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
13716 writing.

13717 CX [EFBIG] An attempt was made to write to a file that exceeds the maximum file size or
13718 the process' file size limit.

13719 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
13720 offset maximum associated with the corresponding stream. |

13721 [EILSEQ] The wide-character code *wc* does not correspond to a valid character. |

13722 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data
13723 was transferred.

13724 CX [EIO] A physical I/O error has occurred, or the process is a member of a
13725 background process group attempting to write to its controlling terminal,
13726 TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the
13727 process group of the process is orphaned. This error may also be returned
13728 under implementation-defined conditions.

13729 CX [ENOSPC] There was no free space remaining on the device containing the file.

13730 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
13731 any process. A SIGPIPE signal shall also be sent to the thread.
```

The *fputwc*() function may fail if:

CX [ENOMEM] Insufficient storage space is available.

CX [ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device. |

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ferror(), *fopen*(), *setbuf*(), *ulimit*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc* is changed from **wint_t** to **wchar_t**.

The Optional Header (OH) marking is removed from **<stdio.h>**.

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EFBIG] and [EIO] mandatory error conditions are added. |
- The [ENOMEM] and [ENXIO] optional error conditions are addd. |

13760 **NAME**

13761 fputws — put a wide-character string on a stream

13762 **SYNOPSIS**

13763 ```
#include <stdio.h>
```

13764 ```
#include <wchar.h>
```

13765 ```
int fputws(const wchar_t *restrict ws, FILE *restrict stream);
```

13766 **DESCRIPTION**

13767 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13768 conflict between the requirements described here and the ISO C standard is unintentional. This
13769 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13770 The *fputws*( ) function shall write a character string corresponding to the (null-terminated)
13771 wide-character string pointed to by *ws* to the stream pointed to by *stream*. No character
13772 corresponding to the terminating null wide-character code shall be written.

13773 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
13774 execution of *fputws*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the
13775 same stream or a call to *exit*( ) or *abort*( ).

13776 **RETURN VALUE**

13777 Upon successful completion, *fputws*( ) shall return a non-negative number. Otherwise, it shall
13778 CX return –1, set an error indicator for the stream, and set *errno* to indicate the error.

13779 **ERRORS**

13780 Refer to *fputwc*( ).

13781 **EXAMPLES**

13782 None.

13783 **APPLICATION USAGE**

13784 The *fputws*( ) function does not append a <newline>.

13785 **RATIONALE**

13786 None.

13787 **FUTURE DIRECTIONS**

13788 None.

13789 **SEE ALSO**

13790 *fopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

13791 **CHANGE HISTORY**

13792 First released in Issue 4. Derived from the MSE working draft.

13793 **Issue 5**

13794 The Optional Header (OH) marking is removed from **<stdio.h>**.

13795 **Issue 6**

13796 Extensions beyond the ISO C standard are now marked.

13797 The *fputws*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.
```

NAME

fread — binary input

SYNOPSIS

```
#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nitems,
    FILE *restrict stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fread*( ) function shall read into the array pointed to by *ptr* up to *nitems* elements whose size is specified by *size* in bytes, from the stream pointed to by *stream*. For each object, *size* calls shall be made to the *fgetc*( ) function and the results stored, in the order read, in an array of **unsigned char** exactly overlaying the object. The file position indicator for the stream (if defined) shall be advanced by the number of bytes successfully read. If an error occurs, the resulting value of the file position indicator for the stream is unspecified. If a partial element is read, its value is unspecified.

CX The *fread*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ), *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

RETURN VALUE

Upon successful completion, *fread*( ) shall return the number of elements successfully read which is less than *nitems* only if a read error or end-of-file is encountered. If *size* or *nitems* is 0, *fread*( ) shall return 0 and the contents of the array and the state of the stream remain unchanged.

CX Otherwise, if a read error occurs, the error indicator for the stream shall be set, and *errno* shall be set to indicate the error.

ERRORS

Refer to *fgetc*( ).

EXAMPLES

Reading from a Stream

The following example reads a single element from the *fp* stream into the array pointed to by *buf*.

```
#include <stdio.h>
...
size_t bytes_read;
char buf[100];
FILE *fp;
...
bytes_read = fread(buf, sizeof(buf), 1, fp);
...
```

APPLICATION USAGE

The *ferror*( ) or *feof*( ) functions must be used to distinguish between an error condition and an end-of-file condition.

Because of possible differences in element length and byte ordering, files written using *fwrite*( ) are application-dependent, and possibly cannot be read using *fread*( ) by a different application

or by the same application on a different processor.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feof( ), *ferror*( ), *fgetc*( ), *fopen*( ), *getc*( ), *gets*( ), *scanf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

- The *fread*( ) prototype is updated.
- The DESCRIPTION is updated to describe how the bytes from a call to *fgetc*( ) are stored.

13858 **NAME**

13859 free — free allocated memory

13860 **SYNOPSIS**

13861 ```
#include <stdlib.h>
```

13862 ```
void free(void *ptr);
```

13863 **DESCRIPTION**

13864 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13865 conflict between the requirements described here and the ISO C standard is unintentional. This
13866 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13867 The *free*( ) function shall cause the space pointed to by *ptr* to be deallocated; that is, made
13868 available for further allocation. If *ptr* is a null pointer, no action shall occur. Otherwise, if the
13869 ADV argument does not match a pointer earlier returned by the *calloc*( ), *malloc*( ), *posix_memalign*( ),
13870 XSI *realloc*( ), or *strdup*( ) function, or if the space has been deallocated by a call to *free*( ) or *realloc*( ),
13871 the behavior is undefined.

13872 Any use of a pointer that refers to freed space results in undefined behavior.

13873 **RETURN VALUE**

13874 The *free*( ) function shall not return a value.

13875 **ERRORS**

13876 No errors are defined.

13877 **EXAMPLES**

13878 None.

13879 **APPLICATION USAGE**

13880 There is now no requirement for the implementation to support the inclusion of **<malloc.h>**.

13881 **RATIONALE**

13882 None.

13883 **FUTURE DIRECTIONS**

13884 None.

13885 **SEE ALSO**

13886 *calloc*( ), *malloc*( ), *realloc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

13887 **CHANGE HISTORY**

13888 First released in Issue 1. Derived from Issue 1 of the SVID.

13889 **Issue 6**

13890 Reference to the *valloc*( ) function is removed.

13891 **NAME**

13892 freeaddrinfo, getaddrinfo — get address information

13893 **SYNOPSIS**

13894 ```
#include <sys/socket.h>
```
13895 ```
#include <netdb.h>
```

13896 ```
void freeaddrinfo(struct addrinfo *ai);
```
13897 ```
int getaddrinfo(const char *restrict nodename,
```
13898 ```
    const char *restrict servname,
```
13899 ```
    const struct addrinfo *restrict hints,
```
13900 ```
    struct addrinfo **restrict res);
```

13901 **DESCRIPTION**

13902 The *freeaddrinfo*( ) function shall free one or more **addrinfo** structures returned by *getaddrinfo*( ), |
13903 along with any additional storage associated with those structures. If the *ai_next* field of the |
13904 structure is not null, the entire list of structures shall be freed. The *freeaddrinfo*( ) function shall |
13905 support the freeing of arbitrary sublists of an **addrinfo** list originally returned by *getaddrinfo*( ).

13906 The *getaddrinfo*( ) function shall translate the name of a service location (for example, a host
13907 name) and/or a service name and shall return a set of socket addresses and associated
13908 information to be used in creating a socket with which to address the specified service.

13909 The *freeaddrinfo*( ) and *getaddrinfo*( ) functions shall be thread-safe.

13910 The *nodename* and *servname* arguments are either null pointers or pointers to null-terminated
13911 strings. One or both of these two arguments shall be supplied by the application as a non-null
13912 pointer.

13913 The format of a valid name depends on the address family or families. If a specific family is not |
13914 given and the name could be interpreted as valid within multiple supported families, the
13915 implementation shall attempt to resolve the name in all supported families and, in absence of
13916 errors, one or more results shall be returned.

13917 If the *nodename* argument is not null, it can be a descriptive name or can be an address string. If
13918 IP6 the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, valid descriptive names
13919 include host names. If the specified address family is AF_INET or AF_UNSPEC, address strings
13920 using Internet standard dot notation as specified in *inet_addr*( ) are valid.

13921 IP6 If the specified address family is AF_INET6 or AF_UNSPEC, standard IPv6 text forms described
13922 in *inet_ntop*( ) are valid.

13923 If *nodename* is not null, the requested service location is named by *nodename*; otherwise, the
13924 requested service location is local to the caller.

13925 If *servname* is null, the call shall return network-level addresses for the specified *nodename*. If
13926 *servname* is not null, it is a null-terminated character string identifying the requested service. This
13927 can be either a descriptive name or a numeric representation suitable for use with the address
13928 IP6 family or families. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, the
13929 service can be specified as a string specifying a decimal port number.

13930 If the *hints* argument is not null, it refers to a structure containing input values that may direct
13931 the operation by providing options and by limiting the returned information to a specific socket
13932 type, address family and/or protocol. In this *hints* structure every member other than *ai_flags*,
13933 *ai_family*, *ai_socktype*, and *ai_protocol* shall be set to zero or a null pointer. A value of
13934 AF_UNSPEC for *ai_family* means that the caller shall accept any address family. A value of zero |
13935 for *ai_socktype* means that the caller shall accept any socket type. A value of zero for *ai_protocol*
13936 means that the caller shall accept any protocol. If *hints* is a null pointer, the behavior shall be as if
```

it referred to a structure containing the value zero for the *ai_flags*, *ai_socktype*, and *ai_protocol* fields, and AF_UNSPEC for the *ai_family* field.

The *ai_flags* field to which the *hints* parameter points shall be set to zero or be the bitwise-inclusive OR of one or more of the values AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST, and AI_NUMERICSERV.

If the AI_PASSIVE flag is specified, the returned address information shall be suitable for use in binding a socket for accepting incoming connections for the specified service. In this case, if the *nodename* argument is null, then the IP address portion of the socket address structure shall be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the AI_PASSIVE flag is not specified, the returned address information shall be suitable for a call to *connect*( ) (for a connection-mode protocol) or for a call to *connect*( ), *sendto*( ), or *sendmsg*( ) (for a connectionless protocol). In this case, if the *nodename* argument is null, then the IP address portion of the socket address structure shall be set to the loopback address.

If the AI_CANONNAME flag is specified and the *nodename* argument is not null, the function shall attempt to determine the canonical name corresponding to *nodename* (for example, if *nodename* is an alias or shorthand notation for a complete name).

If the AI_NUMERICHOST flag is specified, then a non-null *nodename* string supplied shall be a numeric host address string. Otherwise, an [EAI_NONAME] error is returned. This flag shall prevent any type of name resolution service (for example, the DNS) from being invoked.

If the AI_NUMERICSERV flag is specified, then a non-null *servname* string supplied shall be a numeric port string. Otherwise, an [EAI_NONAME] error shall be returned. This flag shall prevent any type of name resolution service (for example, NIS+) from being invoked.

IP6 If the AI_V4MAPPED flag is specified along with an *ai_family* of AF_INET6, then *getaddrinfo*( ) shall return IPv4-mapped IPv6 addresses on finding no matching IPv6 addresses (*ai_addrlen* shall be 16). The AI_V4MAPPED flag shall be ignored unless *ai_family* equals AF_INET6. If the AI_ALL flag is used with the AI_V4MAPPED flag, then *getaddrinfo*( ) shall return all matching IPv6 and IPv4 addresses. The AI_ALL flag without the AI_V4MAPPED flag is ignored.

The *ai_socktype* field to which argument *hints* points specifies the socket type for the service, as defined in *socket*( ). If a specific socket type is not given (for example, a value of zero) and the service name could be interpreted as valid with multiple supported socket types, the implementation shall attempt to resolve the service name for all supported socket types and, in the absence of errors, all possible results shall be returned. A non-zero socket type value shall limit the returned information to values with the specified socket type.

If the *ai_family* field to which *hints* points has the value AF_UNSPEC, addresses shall be returned for use with any address family that can be used with the specified *nodename* and/or *servname*. Otherwise, addresses shall be returned for use only with the specified address family. If *ai_family* is not AF_UNSPEC and *ai_protocol* is not zero, then addresses are returned for use only with the specified address family and protocol; the value of *ai_protocol* shall be interpreted as in a call to the *socket*( ) function with the corresponding values of *ai_family* and *ai_protocol*.

RETURN VALUE

A zero return value for *getaddrinfo*( ) indicates successful completion; a non-zero return value indicates failure. The possible values for the failures are listed in the ERRORS section.

Upon successful return of *getaddrinfo*( ), the location to which *res* points shall refer to a linked list of **addrinfo** structures, each of which shall specify a socket address and information for use in creating a socket with which to use that socket address. The list shall include at least one **addrinfo** structure. The *ai_next* field of each structure contains a pointer to the next structure on the list, or a null pointer if it is the last structure on the list. Each structure on the list shall

include values for use with a call to the *socket*( ) function, and a socket address for use with the *connect*( ) function or, if the AI_PASSIVE flag was specified, for use with the *bind*( ) function. The fields *ai_family*, *ai_socktype*, and *ai_protocol* shall be usable as the arguments to the *socket*( ) function to create a socket suitable for use with the returned address. The fields *ai_addr* and *ai_addrlen* are usable as the arguments to the *connect*( ) or *bind*( ) functions with such a socket, according to the AI_PASSIVE flag.

If *nodename* is not null, and if requested by the AI_CANONNAME flag, the *ai_canonname* field of the first returned **addrinfo** structure shall point to a null-terminated string containing the canonical name corresponding to the input *nodename*; if the canonical name is not available, then *ai_canonname* shall refer to the *nodename* argument or a string with the same contents. The contents of the *ai_flags* field of the returned structures are undefined.

All fields in socket address structures returned by *getaddrinfo*( ) that are not filled in through an explicit argument (for example, *sin6_flowinfo*) shall be set to zero.

Note: This makes it easier to compare socket address structures.

ERRORS

The *getaddrinfo*( ) function shall fail and return the corresponding value if:

[EAI_AGAIN] The name could not be resolved at this time. Future attempts may succeed.

[EAI_BADFLAGS] The *flags* parameter had an invalid value.

[EAI_FAIL] A non-recoverable error occurred when attempting to resolve the name.

[EAI_FAMILY] The address family was not recognized.

[EAI_MEMORY] There was a memory allocation failure when trying to allocate storage for the return value.

[EAI_NONAME] The name does not resolve for the supplied parameters.

Neither *nodename* nor *servname* were supplied. At least one of these shall be supplied.

[EAI_SERVICE] The service passed was not recognized for the specified socket type.

[EAI_SOCKTYPE] The intended socket type was not recognized.

[EAI_SYSTEM] A system error occurred; the error code can be found in *errno*.

[EAI_OVERFLOW] An argument buffer overflowed.

EXAMPLES

None.

APPLICATION USAGE

If the caller handles only TCP and not UDP, for example, then the *ai_protocol* member of the *hints* structure should be set to IPPROTO_TCP when *getaddrinfo*( ) is called.

If the caller handles only IPv4 and not IPv6, then the *ai_family* member of the *hints* structure should be set to AF_INET when *getaddrinfo*( ) is called.

RATIONALE

None.

14024 **FUTURE DIRECTIONS**

14025 None.

14026 **SEE ALSO**

14027 *connect*( ), *gai_strerror*( ), *gethostbyname*( ), *getnameinfo*( ), *getservbyname*( ), *socket*( ), the Base
14028 Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**, **<sys/socket.h>**

14029 **CHANGE HISTORY**

14030 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

14031 The **restrict** keyword is added to the *getaddrinfo*( ) prototype for alignment with the
14032 ISO / IEC 9899: 1999 standard.

14033 **NAME**

14034 freopen — open a stream

14035 **SYNOPSIS**

14036 `#include <stdio.h>`

14037 `FILE *freopen(const char *restrict filename, const char *restrict mode,`
14038 `FILE *restrict stream);`

14039 **DESCRIPTION**

14040 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14041 conflict between the requirements described here and the ISO C standard is unintentional. This
14042 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

14043 The *freopen*( ) function shall first attempt to flush the stream and close any file descriptor
14044 associated with *stream*. Failure to flush or close the file descriptor successfully shall be ignored.
14045 The error and end-of-file indicators for the stream shall be cleared.

14046 The *freopen*( ) function shall open the file whose pathname is the string pointed to by *filename* and |
14047 associate the stream pointed to by *stream* with it. The *mode* argument shall be used just as in
14048 *fopen*( ).

14049 The original stream shall be closed regardless of whether the subsequent open succeeds.

14050 If *filename* is a null pointer, the *freopen*( ) function shall attempt to change the mode of the stream
14051 to that specified by *mode*, as if the name of the file currently associated with the stream had been
14052 used. It is implementation-defined which changes of mode are permitted (if any), and under
14053 what circumstances.

14054 XSI After a successful call to the *freopen*( ) function, the orientation of the stream shall be cleared, the
14055 encoding rule shall be cleared, and the associated **mbstate_t** object shall be set to describe an
14056 initial conversion state.

14057 CX The largest value that can be represented correctly in an object of type **off_t** shall be established
14058 as the offset maximum in the open file description.

14059 **RETURN VALUE**

14060 Upon successful completion, *freopen*( ) shall return the value of *stream*. Otherwise, a null pointer
14061 CX shall be returned, and *errno* shall be set to indicate the error.

14062 **ERRORS**

14063 The *freopen*( ) function shall fail if:

14064 CX [EACCES] Search permission is denied on a component of the path prefix, or the file
14065 exists and the permissions specified by *mode* are denied, or the file does not
14066 exist and write permission is denied for the parent directory of the file to be
14067 created.

14068 CX [EINTR] A signal was caught during *freopen*( ).

14069 CX [EISDIR] The named file is a directory and *mode* requires write access.

14070 CX [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
14071 argument.

14072 CX [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

14073 CX [ENAMETOOLONG]
14074 The length of the *filename* argument exceeds {PATH_MAX} or a pathname |
14075 component is longer than {NAME_MAX}. |

14076 CX [ENFILE] The maximum allowable number of files is currently open in the system.

14077 CX [ENOENT] A component of *filename* does not name an existing file or *filename* is an empty
14078 string.

14079 CX [ENOSPC] The directory or file system that would contain the new file cannot be
14080 expanded, the file does not exist, and it was to be created.

14081 CX [ENOTDIR] A component of the path prefix is not a directory.

14082 CX [ENXIO] The named file is a character special or block special file, and the device
14083 associated with this special file does not exist.

14084 CX [EOVERFLOW] The named file is a regular file and the size of the file cannot be represented
14085 correctly in an object of type **off_t**.

14086 CX [EROFS] The named file resides on a read-only file system and *mode* requires write
14087 access.

14088 The *freopen*( ) function may fail if:

14089 CX [EINVAL] The value of the *mode* argument is not valid.

14090 CX [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
14091 resolution of the *path* argument.

14092 CX [ENAMETOOLONG]
14093 Pathname resolution of a symbolic link produced an intermediate result |
14094 whose length exceeds {PATH_MAX}.

14095 CX [ENOMEM] Insufficient storage space is available.

14096 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
14097 capabilities of the device.

14098 CX [ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *mode*
14099 requires write access.

14100 **EXAMPLES**

14101 **Directing Standard Output to a File**

14102 The following example logs all standard output to the **/tmp/logfile** file.

```
14103 #include <stdio.h>
14104 ...
14105 FILE *fp;
14106 ...
14107 fp = freopen ("/tmp/logfile", "a+", stdout);
14108 ...
```

14109 **APPLICATION USAGE**

14110 The *freopen*( ) function is typically used to attach the preopened *streams* associated with *stdin*,
14111 *stdout*, and *stderr* to other files.

14112 **RATIONALE**

14113 None.

14114 **FUTURE DIRECTIONS**

14115 None.

14116 **SEE ALSO**

14117 *fclose*( ), *fopen*( ), *fdopen*( ), *mbsinit*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
14118 **<stdio.h>**

14119 **CHANGE HISTORY**

14120 First released in Issue 1. Derived from Issue 1 of the SVID.

14121 **Issue 5**

14122 The DESCRIPTION is updated to indicate that the orientation of the stream is cleared and the
14123 conversion state of the stream is set to an initial conversion state by a successful call to the
14124 *freopen*( ) function.

14125 Large File Summit extensions are added.

14126 **Issue 6**

14127 Extensions beyond the ISO C standard are now marked.

14128 The following new requirements on POSIX implementations derive from alignment with the |
14129 Single UNIX Specification:

14130 • In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file
14131 description. This change is to support large files.

14132 • In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support
14133 large files.

14134 • The [ELOOP] mandatory error condition is added.

14135 • A second [ENAMETOOLONG] is added as an optional error condition.

14136 • The [EINVAL], [ENOMEM], [ENXIO], and [ETXTBSY] optional error conditions are added.

14137 The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

14138 • The *freopen*( ) prototype is updated.

14139 • The DESCRIPTION is updated.

14140 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
14141 [ELOOP] error condition is added.

14142 The DESCRIPTION is updated regarding failure to close, changing the ‘‘file’’ to ‘‘file descriptor’’.

14143 **NAME**

14144 frexp, frexpf, frexpl — extract mantissa and exponent from a double precision number

14145 **SYNOPSIS**

14146
```
#include <math.h>
```

14147
```
double frexp(double num, int *exp);
```
14148
```
float frexpf(float num, int *exp);
```
14149
```
long double frexpl(long double num, int *exp);
```

14150 **DESCRIPTION**

14151 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14152 conflict between the requirements described here and the ISO C standard is unintentional. This
14153 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

14154 These functions shall break a floating-point number *num* into a normalized fraction and an |
14155 integral power of 2. The integer exponent shall be stored in the **int** object pointed to by *exp*. |

14156 **RETURN VALUE**

14157 For finite arguments, these functions shall return the value *x*, such that *x* has a magnitude in the
14158 interval [½,1) or 0, and *num* equals *x* times 2 raised to the power **exp*.

14159 MX If *num* is NaN, a NaN shall be returned, and the value of **exp* is unspecified.

14160 If *num* is ±0, ±0 shall be returned, and the value of **exp* shall be 0.

14161 If *num* is ±Inf, *num* shall be returned, and the value of **exp* is unspecified.

14162 **ERRORS**

14163 No errors are defined.

14164 **EXAMPLES**

14165 None.

14166 **APPLICATION USAGE**

14167 None.

14168 **RATIONALE**

14169 None.

14170 **FUTURE DIRECTIONS**

14171 None.

14172 **SEE ALSO**

14173 *isnan*( ), *ldexp*( ), *modf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

14174 **CHANGE HISTORY**

14175 First released in Issue 1. Derived from Issue 1 of the SVID.

14176 **Issue 5**

14177 The DESCRIPTION is updated to indicate how an application should check for an error. This
14178 text was previously published in the APPLICATION USAGE section.

14179 **Issue 6**

14180 The *frexpf*( ) and *frexpl*( ) functions are added for alignment with the ISO/IEC 9899:1999
14181 standard. |

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard. |

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

14186 **NAME**

14187 fscanf, scanf, sscanf — convert formatted input

14188 **SYNOPSIS**

14189 `#include <stdio.h>`

14190 `int fscanf(FILE *restrict stream, const char *restrict format, ... );`
14191 `int scanf(const char *restrict format, ... );`
14192 `int sscanf(const char *restrict s, const char *restrict format, ... );`

14193 **DESCRIPTION**

14194 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14195 conflict between the requirements described here and the ISO C standard is unintentional. This
14196 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

14197 The *fscanf*() function shall read from the named input *stream*. The *scanf*() function shall read
14198 from the standard input stream *stdin*. The *sscanf*() function shall read from the string *s*. Each
14199 function reads bytes, interprets them according to a format, and stores the results in its
14200 arguments. Each expects, as arguments, a control string *format* described below, and a set of
14201 *pointer* arguments indicating where the converted input should be stored. The result is
14202 undefined if there are insufficient arguments for the format. If the format is exhausted while
14203 arguments remain, the excess arguments shall be evaluated but otherwise ignored.

14204 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
14205 to the next unused argument. In this case, the conversion specifier character % (see below) is
14206 replaced by the sequence "%*n*$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}].
14207 This feature provides for the definition of format strings that select arguments in an order
14208 appropriate to specific languages. In format strings containing the "%*n*$" form of conversion
14209 specifications, it is unspecified whether numbered arguments in the argument list can be
14210 referenced from the format string more than once.

14211 The *format* can contain either form of a conversion specification—that is, % or "%*n*$"—but the
14212 two forms cannot be mixed within a single *format* string. The only exception to this is that %% or
14213 %* can be mixed with the "%*n*$" form. When numbered argument specifications are used,
14214 specifying the *N*th argument requires that all the leading arguments, from the first to the
14215 (*N*−1)th, are pointers.

14216 CX The *fscanf*() function in all its forms shall allow detection of a language-dependent radix
14217 character in the input string. The radix character is defined in the program's locale (category
14218 *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the
14219 radix character shall default to a period ('.').

14220 The format is a character string, beginning and ending in its initial shift state, if any, composed
14221 of zero or more directives. Each directive is composed of one of the following: one or more
14222 white-space characters (<space>s, <tab>s, <newline>s, <vertical-tab>s, or <form-feed>s); an
14223 ordinary character (neither '%' nor a white-space character); or a conversion specification. Each
14224 XSI conversion specification is introduced by the character '%' or the character sequence "%*n*$",
14225 after which the following appear in sequence:

14226 - An optional assignment-suppressing character '*'.
14227 - An optional non-zero decimal integer that specifies the maximum field width.
14228 - An option length modifier that specifies the size of the receiving object.
14229 - A *conversion specifier* character that specifies the type of conversion to be applied. The valid
14230 conversion specifiers are described below.

The *fscanf*( ) functions shall execute each directive of the format in turn. If a directive fails, as detailed below, the function shall return. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space characters shall be executed by reading input until no more valid input can be read, or up to the first byte which is not a white-space character, which remains unread.

A directive that is an ordinary character shall be executed as follows: the next byte shall be read from the input and compared with the byte that comprises the directive; if the comparison shows that they are not equivalent, the directive shall fail, and the differing and subsequent bytes shall remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive shall fail.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion character. A conversion specification shall be executed in the following steps.

Input white-space characters (as specified by *isspace*( )) shall be skipped, unless the conversion specification includes a `[`, `c`, `C`, or `n` conversion specifier.

An item shall be read from the input, unless the conversion specification includes an `n` conversion specifier. An input item shall be defined as the longest sequence of input bytes (up to any specified maximum field width, which may be measured in characters or bytes dependent on the conversion specifier) which is an initial subsequence of a matching sequence. The first byte, if any, after the input item shall remain unread. If the length of the input item is 0, the execution of the conversion specification shall fail; this condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a `%` conversion specifier, the input item (or, in the case of a `%n` conversion specification, the count of input bytes) shall be converted to a type appropriate to the conversion character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure. Unless assignment suppression was indicated by a `'*'`, the result of the conversion shall be placed in the object pointed to by the first argument following the *format* argument that has not already received a conversion result if the conversion specification is introduced by `%`, or in the *n*th argument if introduced by the character sequence `"%`*n*`$"`. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

XSI

The length modifiers and their meanings are:

`hh` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.

`h` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **short** or **unsigned short**.

`l` (ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **long** or **unsigned long**; that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to **double**; or that a following `c`, `s`, or `[` conversion specifier applies to an argument with type pointer to **wchar_t**.

`ll` (ell-ell)
Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **long long** or **unsigned long long**.

14277 j Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
14278 argument with type pointer to **intmax_t** or **uintmax_t**.

14279 z Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
14280 argument with type pointer to **size_t** or the corresponding signed integer type.

14281 t Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
14282 argument with type pointer to **ptrdiff_t** or the corresponding **unsigned** type.

14283 L Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an
14284 argument with type pointer to **long double**.

14285 If a length modifier appears with any conversion specifier other than as specified above, the
14286 behavior is undefined.

14287 The following conversion specifiers are valid:

14288 d Matches an optionally signed decimal integer, whose format is the same as expected for
14289 the subject sequence of *strtol*( ) with the value 10 for the *base* argument. In the absence
14290 of a size modifier, the application shall ensure that the corresponding argument is a
14291 pointer to **int**.

14292 i Matches an optionally signed integer, whose format is the same as expected for the
14293 subject sequence of *strtol*( ) with 0 for the *base* argument. In the absence of a size
14294 modifier, the application shall ensure that the corresponding argument is a pointer to
14295 **int**.

14296 o Matches an optionally signed octal integer, whose format is the same as expected for
14297 the subject sequence of *strtoul*( ) with the value 8 for the *base* argument. In the absence
14298 of a size modifier, the application shall ensure that the corresponding argument is a
14299 pointer to **unsigned**.

14300 u Matches an optionally signed decimal integer, whose format is the same as expected for
14301 the subject sequence of *strtoul*( ) with the value 10 for the *base* argument. In the absence
14302 of a size modifier, the application shall ensure that the corresponding argument is a
14303 pointer to **unsigned**.

14304 x Matches an optionally signed hexadecimal integer, whose format is the same as
14305 expected for the subject sequence of *strtoul*( ) with the value 16 for the *base* argument. In
14306 the absence of a size modifier, the application shall ensure that the corresponding
14307 argument is a pointer to **unsigned**.

14308 a, e, f, g
14309 Matches an optionally signed floating-point number, infinity, or NaN, whose format is
14310 the same as expected for the subject sequence of *strtod*( ). In the absence of a size
14311 modifier, the application shall ensure that the corresponding argument is a pointer to
14312 **float**.

14313 If the *fprintf*( ) family of functions generates character string representations for infinity
14314 and NaN (a symbolic entity encoded in floating-point format) to support
14315 IEEE Std 754-1985, the *fscanf*( ) family of functions shall recognize them as input.

14316 s Matches a sequence of bytes that are not white-space characters. The application shall
14317 ensure that the corresponding argument is a pointer to the initial byte of an array of
14318 **char**, **signed char**, or **unsigned char** large enough to accept the sequence and a
14319 terminating null character code, which shall be added automatically.

14320 If an l (ell) qualifier is present, the input is a sequence of characters that begins in the |
14321 initial shift state. Each character shall be converted to a wide character as if by a call to |

the *mbrtowc*() function, with the conversion state described by an **mbstate_t** object initialized to zero before the first character is converted. The application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically.

[Matches a non-empty sequence of bytes from a set of expected bytes (the *scanset*). The normal skip over white-space characters shall be suppressed in this case. The application shall ensure that the corresponding argument is a pointer to the initial byte of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence and a terminating null byte, which shall be added automatically.

If an `l` (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character in the sequence shall be converted to a wide character as if by a call to the *mbrtowc*() function, with the conversion state described by an **mbstate_t** object initialized to zero before the first character is converted. The application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically.

The conversion specification includes all subsequent bytes in the *format* string up to and including the matching right square bracket (`']'`). The bytes between the square brackets (the *scanlist*) comprise the scanset, unless the byte after the left square bracket is a circumflex (`'^'`), in which case the scanset contains all bytes that do not appear in the scanlist between the circumflex and the right square bracket. If the conversion specification begins with `"[]"` or `"[^]"`, the right square bracket is included in the scanlist and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise, the first right square bracket is the one that ends the conversion specification. If a `'-'` is in the scanlist and is not the first character, nor the second where the first character is a `'^'`, nor the last character, the behavior is implementation-defined.

c Matches a sequence of bytes of the number specified by the field width (1 if no field width is present in the conversion specification). The application shall ensure that the corresponding argument is a pointer to the initial byte of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence. No null byte is added. The normal skip over white-space characters shall be suppressed in this case.

If an `l` (ell) qualifier is present, the input shall be a sequence of characters that begins in the initial shift state. Each character in the sequence is converted to a wide character as if by a call to the *mbrtowc*() function, with the conversion state described by an **mbstate_t** object initialized to zero before the first character is converted. The application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the resulting sequence of wide characters. No null wide character is added.

p Matches an implementation-defined set of sequences, which shall be the same as the set of sequences that is produced by the %p conversion specification of the corresponding *fprintf*() functions. The application shall ensure that the corresponding argument is a pointer to a pointer to **void**. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise, the behavior of the %p conversion specification is undefined.

n No input is consumed. The application shall ensure that the corresponding argument is a pointer to the integer into which shall be written the number of bytes read from the

input so far by this call to the *fscanf*() functions. Execution of a %n conversion specification shall not increment the assignment count returned at the completion of execution of the function. No argument shall be converted, but one shall be consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

XSI C Equivalent to lc.

XSI S Equivalent to ls.

% Matches a single '%' character; no conversion or assignment occurs. The complete conversion specification shall be %%.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers A, E, F, G, and X are also valid and shall be equivalent to a, e, f, g, and x, respectively.

If end-of-file is encountered during input, conversion shall be terminated. If end-of-file occurs before any bytes matching the current conversion specification (except for %n) have been read (other than leading white-space characters, where permitted), execution of the current conversion specification shall terminate with an input failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if any) shall be terminated with an input failure.

Reaching the end of the string in *sscanf*() shall be equivalent to encountering end-of-file for *fscanf*().

If conversion terminates on a conflicting input, the offending input is left unread in the input. Any trailing white space (including <newline>s) shall be left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the %n conversion specification.

CX The *fscanf*() and *scanf*() functions may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc*(), *fgets*(), *fread*(), *getc*(), *getchar*(), *gets*(), *fscanf*(), or *fscanf*() using *stream* that returns data not supplied by a prior call to *ungetc*().

RETURN VALUE

Upon successful completion, these functions shall return the number of successfully matched and assigned input items; this number can be zero in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF shall be returned. If a read CX error occurs, the error indicator for the stream is set, EOF shall be returned, and *errno* shall be set to indicate the error.

ERRORS

For the conditions under which the *fscanf*() functions fail and may fail, refer to *fgetc*() or *fgetwc*().

In addition, *fscanf*() may fail if:

XSI [EILSEQ] Input byte sequence does not form a valid character.

XSI [EINVAL] There are insufficient arguments.

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 Hamster
```

assigns to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* contains the string `"Hamster"`.

The call:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

assigns 56 to *i*, 789.0 to *x*, skips 0123, and places the string `"56\0"` in *name*. The next call to *getchar*( ) shall return the character `'a'`.

Reading Data into an Array

The following call uses *fscanf*( ) to read three floating-point numbers from standard input into the *input* array.

```
float input[3]; fscanf (stdin, "%f %f %f", input, input+1, input+2);
```

APPLICATION USAGE

If the application calling *fscanf*( ) has any objects of type **wint_t** or **wchar_t**, it must also include the **<wchar.h>** header to have these objects defined.

RATIONALE

This function is aligned with the ISO/IEC 9899:1999 standard, and in doing so a few ''obvious'' things were not included. Specifically, the set of characters allowed in a scanset is limited to single-byte characters. In other similar places, multi-byte characters have been permitted, but for alignment with the ISO/IEC 9899:1999 standard, it has not been done here. Applications needing this could use the corresponding wide-character functions to achieve the desired results.

FUTURE DIRECTIONS

None.

SEE ALSO

getc( ), *printf*( ), *setlocale*( ), *strtod*( ), *strtol*( ), *strtoul*( ), *wcrtomb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<langinfo.h>**, **<stdio.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the `l` (ell) qualifier is now defined for the `c`, `s`, and `[` conversion specifiers.

The DESCRIPTION is updated to indicate that if infinity and NaN can be generated by the *fprintf*( ) family of functions, then they are recognized by the *fscanf*( ) family.

Issue 6

The Open Group Corrigendum U021/7 and U028/10 are applied. These correct several occurrences of ‘‘characters’’ in the text which have been replaced with the term ‘‘bytes’’.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The prototypes for *fscanf*(), *scanf*(), and *sscanf*() are updated.
- The DESCRIPTION is updated.
- The `hh`, `ll`, `j`, `t`, and `z` length modifiers are added.
- The `a`, `A`, and `F` conversion characters are added.

The DESCRIPTION is updated to use the terms ‘‘conversion specifier’’ and ‘‘conversion specification’’ consistently.

14464 **NAME**

14465 fseek, fseeko — reposition a file-position indicator in a stream

14466 **SYNOPSIS**

14467 `#include <stdio.h>`

14468 `int fseek(FILE *stream, long offset, int whence);`

14469 CX `int fseeko(FILE *stream, off_t offset, int whence);`

14470

14471 **DESCRIPTION**

14472 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14473 conflict between the requirements described here and the ISO C standard is unintentional. This
14474 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

14475 The *fseek*( ) function shall set the file-position indicator for the stream pointed to by *stream*. If a
14476 read or write error occurs, the error indicator for the stream shall be set and *fseek*( ) fails.

14477 The new position, measured in bytes from the beginning of the file, shall be obtained by adding
14478 *offset* to the position specified by *whence*. The specified point is the beginning of the file for
14479 SEEK_SET, the current value of the file-position indicator for SEEK_CUR, or end-of-file for
14480 SEEK_END.

14481 If the stream is to be used with wide-character input/output functions, the application shall
14482 ensure that *offset* is either 0 or a value returned by an earlier call to *ftell*( ) on the same stream and
14483 *whence* is SEEK_SET.

14484 A successful call to *fseek*( ) shall clear the end-of-file indicator for the stream and undo any effects
14485 of *ungetc*( ) and *ungetwc*( ) on the same stream. After an *fseek*( ) call, the next operation on an
14486 update stream may be either input or output.

14487 CX If the most recent operation, other than *ftell*( ), on a given stream is *fflush*( ), the file offset in the
14488 underlying open file description shall be adjusted to reflect the location specified by *fseek*( ).

14489 The *fseek*( ) function shall allow the file-position indicator to be set beyond the end of existing
14490 data in the file. If data is later written at this point, subsequent reads of data in the gap shall
14491 return bytes with the value 0 until data is actually written into the gap.

14492 The behavior of *fseek*( ) on devices which are incapable of seeking is implementation-defined.
14493 The value of the file offset associated with such a device is undefined.

14494 If the stream is writable and buffered data had not been written to the underlying file, *fseek*( )
14495 shall cause the unwritten data to be written to the file and shall mark the *st_ctime* and *st_mtime*
14496 fields of the file for update.

14497 In a locale with state-dependent encoding, whether *fseek*( ) restores the stream's shift state is
14498 implementation-defined.

14499 The *fseeko*( ) function shall be equivalent to the *fseek*( ) function except that the *offset* argument is |
14500 of type **off_t**.

14501 **RETURN VALUE**

14502 CX The *fseek*( ) and *fseeko*( ) functions shall return 0 if they succeed.

14503 CX Otherwise, they shall return –1 and set *errno* to indicate the error.

14504 **ERRORS**

14505 CX The *fseek*( ) and *fseeko*( ) functions shall fail if, either the *stream* is unbuffered or the *stream*'s
14506 buffer needed to be flushed, and the call to *fseek*( ) or *fseeko*( ) causes an underlying *lseek*( ) or
14507 *write*( ) to be invoked, and:

14508	CX	[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the write operation.
14510	CX	[EBADF]	The file descriptor underlying the stream file is not open for writing or the stream's buffer needed to be flushed and the file is not open.
14512	CX	[EFBIG]	An attempt was made to write a file that exceeds the maximum file size.
14513	XSI	[EFBIG]	An attempt was made to write a file that exceeds the process' file size limit.
14514	CX	[EFBIG]	The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
14516	CX	[EINTR]	The write operation was terminated due to the receipt of a signal, and no data was transferred.
14518	CX	[EINVAL]	The *whence* argument is invalid. The resulting file-position indicator would be set to a negative value.
14520	CX	[EIO]	A physical I/O error has occurred, or the process is a member of a background process group attempting to perform a *write*() to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.
14525	CX	[ENOSPC]	There was no free space remaining on the device containing the file.
14526	CX	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
14528	CX	[EOVERFLOW]	For *fseek*(), the resulting file offset would be a value which cannot be represented correctly in an object of type **long**.
14530	CX	[EOVERFLOW]	For *fseeko*(), the resulting file offset would be a value which cannot be represented correctly in an object of type **off_t**.
14532	CX	[EPIPE]	An attempt was made to write to a pipe or FIFO that is not open for reading by any process; a SIGPIPE signal shall also be sent to the thread.
14534	CX	[ESPIPE]	The file descriptor underlying *stream* is associated with a pipe or FIFO.

14535 **EXAMPLES**

14536 None.

14537 **APPLICATION USAGE**

14538 None.

14539 **RATIONALE**

14540 None.

14541 **FUTURE DIRECTIONS**

14542 None.

14543 **SEE ALSO**

14544 *fopen*(), *fsetpos*(), *ftell*(), *getrlimit*(), *lseek*(), *rewind*(), *ulimit*(), *ungetc*(), *write*(), the Base
14545 Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

14546 **CHANGE HISTORY**

14547 First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The *fseeko*( ) function is added.
- The [EFBIG], [EOVERFLOW], and [ENXIO] mandatory error conditions are added.

The following change is incorporated for alignment with the FIPS requirements:

- The [EINTR] error is no longer an indication that the implementation does not report partial transfers.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The DESCRIPTION is updated to explicitly state that *fseek*( ) sets the file-position indicator, and then on error the error indicator is set and *fseek*( ) fails. This is for alignment with the ISO / IEC 9899: 1999 standard.

14565 **NAME**

14566 fsetpos — set current file position

14567 **SYNOPSIS**

14568 `#include <stdio.h>`

14569 `int fsetpos(FILE *stream, const fpos_t *pos);`

14570 **DESCRIPTION**

14571 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14572 conflict between the requirements described here and the ISO C standard is unintentional. This
14573 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

14574 The *fsetpos*( ) function shall set the file position and state indicators for the stream pointed to by
14575 *stream* according to the value of the object pointed to by *pos*, which the application shall ensure
14576 is a value obtained from an earlier call to *fgetpos*( ) on the same stream. If a read or write error
14577 occurs, the error indicator for the stream shall be set and *fsetpos*( ) fails.

14578 A successful call to the *fsetpos*( ) function shall clear the end-of-file indicator for the stream and
14579 undo any effects of *ungetc*( ) on the same stream. After an *fsetpos*( ) call, the next operation on an
14580 update stream may be either input or output.

14581 CX The behavior of *fsetpos*( ) on devices which are incapable of seeking is implementation-defined.
14582 The value of the file offset associated with such a device is undefined.

14583 **RETURN VALUE**

14584 The *fsetpos*( ) function shall return 0 if it succeeds; otherwise, it shall return a non-zero value and
14585 set *errno* to indicate the error.

14586 **ERRORS**

14587 CX The *fsetpos*( ) function shall fail if, either the *stream* is unbuffered or the *stream*'s buffer needed to
14588 be flushed, and the call to *fsetpos*( ) causes an underlying *lseek*( ) or *write*( ) to be invoked, and:

14589 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the process would be
14590 delayed in the write operation.

14591 CX [EBADF] The file descriptor underlying the stream file is not open for writing or the
14592 stream's buffer needed to be flushed and the file is not open.

14593 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

14594 XSI [EFBIG] An attempt was made to write a file that exceeds the process' file size limit.

14595 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
14596 offset maximum associated with the corresponding stream.

14597 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data
14598 was transferred.

14599 CX [EINVAL] The *whence* argument is invalid. The resulting file-position indicator would be
14600 set to a negative value.

14601 CX [EIO] A physical I/O error has occurred, or the process is a member of a
14602 background process group attempting to perform a *write*( ) to its controlling
14603 terminal, TOSTOP is set, the process is neither ignoring nor blocking
14604 SIGTTOU, and the process group of the process is orphaned. This error may
14605 also be returned under implementation-defined conditions.

14606 CX [ENOSPC] There was no free space remaining on the device containing the file.

14607 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
14608 capabilities of the device.

14609 CX [EPIPE] The file descriptor underlying *stream* is associated with a pipe or FIFO.

14610 CX [EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading
14611 by any process; a SIGPIPE signal shall also be sent to the thread.

14612 **EXAMPLES**

14613 None.

14614 **APPLICATION USAGE**

14615 None.

14616 **RATIONALE**

14617 None.

14618 **FUTURE DIRECTIONS**

14619 None.

14620 **SEE ALSO**

14621 *fopen*( ), *ftell*( ), *lseek*( ), *rewind*( ), *ungetc*( ), *write*( ), the Base Definitions volume of
14622 IEEE Std 1003.1-200x, **<stdio.h>**

14623 **CHANGE HISTORY**

14624 First released in Issue 4. Derived from the ISO C standard.

14625 **Issue 6**

14626 Extensions beyond the ISO C standard are now marked.

14627 An additional [ESPIPE] error condition is added for sockets.

14628 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

14629 The DESCRIPTION is updated to clarify that the error indicator is set for the stream on a read or
14630 write error. This is for alignment with the ISO / IEC 9899: 1999 standard.

14631 **NAME**

14632 fstat — get file status

14633 **SYNOPSIS**

14634 `#include <sys/stat.h>`

14635 `int fstat(int fildes, struct stat *buf);`

14636 **DESCRIPTION**

14637 The *fstat*( ) function shall obtain information about an open file associated with the file
14638 descriptor *fildes*, and shall write it to the area pointed to by *buf*.

14639 SHM If *fildes* references a shared memory object, the implementation shall update in the **stat** structure
14640 pointed to by the *buf* argument only the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the
14641 S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be
14642 valid. The implementation may update other fields and flags.

14643 TYM If *fildes* references a typed memory object, the implementation shall update in the **stat** structure
14644 pointed to by the *buf* argument only the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the
14645 S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be
14646 valid. The implementation may update other fields and flags.

14647 The *buf* argument is a pointer to a **stat** structure, as defined in **<sys/stat.h>**, into which
14648 information is placed concerning the file.

14649 The structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime*
14650 shall have meaningful values for all other file types defined in this volume of
14651 IEEE Std 1003.1-200x. The value of the member *st_nlink* shall be set to the number of links to the
14652 file.

14653 An implementation that provides additional or alternative file access control mechanisms may,
14654 under implementation-defined conditions, cause *fstat*( ) to fail.

14655 The *fstat*( ) function shall update any time-related fields as described in the Base Definitions
14656 volume of IEEE Std 1003.1-200x, Section 4.7, File Times Update, before writing into the **stat**
14657 structure.

14658 **RETURN VALUE**

14659 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
14660 indicate the error.

14661 **ERRORS**

14662 The *fstat*( ) function shall fail if:

14663 [EBADF] The *fildes* argument is not a valid file descriptor.

14664 [EIO] An I/O error occurred while reading from the file system.

14665 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file
14666 serial number cannot be represented correctly in the structure pointed to by
14667 *buf*.

14668 The *fstat*( ) function may fail if:

14669 [EOVERFLOW] One of the values is too large to store into the structure pointed to by the *buf*
14670 argument.

EXAMPLES

Obtaining File Status Information

The following example shows how to obtain file status information for a file named **/home/cnd/mod1**. The structure variable *buffer* is defined for the **stat** structure. The **/home/cnd/mod1** file is opened with read/write privileges and is passed to the open file descriptor *fildes*.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

struct stat buffer;
int         status;
...
fildes = open("/home/cnd/mod1", O_RDWR);
status = fstat(fildes, &buffer);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

lstat( ), *stat*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [EIO] mandatory error condition is added.
- The [EOVERFLOW] mandatory error condition is added. This change is to support large files.
- The [EOVERFLOW] optional error condition is added.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that shared memory object semantics apply to typed memory objects.

14711 **NAME**

14712 fstatvfs, statvfs — get file system information

14713 **SYNOPSIS**

14714 XSI
```
#include <sys/statvfs.h>

int fstatvfs(int fildes, struct statvfs *buf);
int statvfs(const char *restrict path, struct statvfs *restrict buf);
```
14717

14718 **DESCRIPTION**

14719 The *fstatvfs*( ) function shall obtain information about the file system containing the file |
14720 referenced by *fildes*. |

14721 The *statvfs*( ) function shall obtain information about the file system containing the file named by |
14722 *path*. |

14723 For both functions, the *buf* argument is a pointer to a **statvfs** structure that shall be filled. Read, |
14724 write, or execute permission of the named file is not required. |

14725 The following flags can be returned in the *f_flag* member: |

14726 ST_RDONLY Read-only file system.

14727 ST_NOSUID Setuid/setgid bits ignored by *exec*.

14728 It is unspecified whether all members of the **statvfs** structure have meaningful values on all file |
14729 systems.

14730 **RETURN VALUE**

14731 Upon successful completion, *statvfs*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to
14732 indicate the error.

14733 **ERRORS**

14734 The *fstatvfs*( ) and *statvfs*( ) functions shall fail if:

14735 [EIO] An I/O error occurred while reading the file system.

14736 [EINTR] A signal was caught during execution of the function.

14737 [EOVERFLOW] One of the values to be returned cannot be represented correctly in the
14738 structure pointed to by *buf*.

14739 The *fstatvfs*( ) function shall fail if:

14740 [EBADF] The *fildes* argument is not an open file descriptor.

14741 The *statvfs*( ) function shall fail if:

14742 [EACCES] Search permission is denied on a component of the path prefix.

14743 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
14744 argument.

14745 [ENAMETOOLONG]
14746 The length of a pathname exceeds {PATH_MAX} or a pathname component is |
14747 longer than {NAME_MAX}. |

14748 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

14749 [ENOTDIR] A component of the path prefix of *path* is not a directory.

14750 The *statvfs*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

EXAMPLES

Obtaining File System Information Using fstatvfs()

The following example shows how to obtain file system information for the file system upon which the file named **/home/cnd/mod1** resides, using the *fstatvfs*() function. The **/home/cnd/mod1** file is opened with read/write privileges and the open file descriptor is passed to the *fstatvfs*() function.

```
#include <statvfs.h>
#include <fcntl.h>

struct statvfs buffer;
int            status;
...
fildes = open("/home/cnd/mod1", O_RDWR);
status  = fstatvfs(fildes, &buffer);
```

Obtaining File System Information Using statvfs()

The following example shows how to obtain file system information for the file system upon which the file named **/home/cnd/mod1** resides, using the *statvfs*() function.

```
#include <statvfs.h>

struct statvfs buffer;
int            status;
...
status = statvfs("/home/cnd/mod1", &buffer);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod(), *chown*(), *creat*(), *dup*(), *exec*, *fcntl*(), *link*(), *mknod*(), *open*(), *pipe*(), *read*(), *time*(), *unlink*(), *utime*(), *write*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/statvfs.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Large File Summit extensions are added.

14791 **Issue 6**

14792 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

14793 The **restrict** keyword is added to the *statvfs*( ) prototype for alignment with the
14794 ISO / IEC 9899: 1999 standard.

14795 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
14796 [ELOOP] error condition is added.

14797 **NAME**

14798 fsync — synchronize changes to a file

14799 **SYNOPSIS**

14800 FSC
```
#include <unistd.h>

int fsync(int fildes);
```
14801

14802

14803 **DESCRIPTION**

14804 The *fsync*( ) function shall request that all data for the open file descriptor named by *fildes* is to be
14805 transferred to the storage device associated with the file described by *fildes* in an
14806 implementation-defined manner. The *fsync*( ) function shall not return until the system has
14807 completed that action or until an error is detected.

14808 SIO If _POSIX_SYNCHRONIZED_IO is defined, the *fsync*( ) function shall force all currently queued
14809 I/O operations associated with the file indicated by file descriptor *fildes* to the synchronized I/O
14810 completion state. All I/O operations shall be completed as defined for synchronized I/O file
14811 integrity completion.

14812 **RETURN VALUE**

14813 Upon successful completion, *fsync*( ) shall return 0. Otherwise, –1 shall be returned and *errno* set
14814 to indicate the error. If the *fsync*( ) function fails, outstanding I/O operations are not guaranteed
14815 to have been completed.

14816 **ERRORS**

14817 The *fsync*( ) function shall fail if:

14818 [EBADF] The *fildes* argument is not a valid descriptor.

14819 [EINTR] The *fsync*( ) function was interrupted by a signal.

14820 [EINVAL] The *fildes* argument does not refer to a file on which this operation is possible.

14821 [EIO] An I/O error occurred while reading from or writing to the file system.

14822 In the event that any of the queued I/O operations fail, *fsync*( ) shall return the error conditions
14823 defined for *read*( ) and *write*( ).

14824 **EXAMPLES**

14825 None.

14826 **APPLICATION USAGE**

14827 The *fsync*( ) function should be used by programs which require modifications to a file to be
14828 completed before continuing; for example, a program which contains a simple transaction
14829 facility might use it to ensure that all modifications to a file or files caused by a transaction are
14830 recorded.

14831 **RATIONALE**

14832 The *fsync*( ) function is intended to force a physical write of data from the buffer cache, and to
14833 assure that after a system crash or other failure that all data up to the time of the *fsync*( ) call is
14834 recorded on the disk. Since the concepts of ‘‘buffer cache’’, ‘‘system crash’’, ‘‘physical write’’, and
14835 ‘‘non-volatile storage’’ are not defined here, the wording has to be more abstract.

14836 If _POSIX_SYNCHRONIZED_IO is not defined, the wording relies heavily on the conformance
14837 document to tell the user what can be expected from the system. It is explicitly intended that a
14838 null implementation is permitted. This could be valid in the case where the system cannot assure
14839 non-volatile storage under any circumstances or when the system is highly fault-tolerant and the
14840 functionality is not required. In the middle ground between these extremes, *fsync*( ) might or
14841 might not actually cause data to be written where it is safe from a power failure. The

conformance document should identify at least that one configuration exists (and how to obtain that configuration) where this can be assured for at least some files that the user can select to use for critical data. It is not intended that an exhaustive list is required, but rather sufficient information is provided to let the user determine that if he or she has critical data he or she can configure her system to allow it to be written to non-volatile storage.

It is reasonable to assert that the key aspects of *fsync*( ) are unreasonable to test in a test suite. That does not make the function any less valuable, just more difficult to test. A formal conformance test should probably force a system crash (power shutdown) during the test for this condition, but it needs to be done in such a way that automated testing does not require this to be done except when a formal record of the results is being made. It would also not be unreasonable to omit testing for *fsync*( ), allowing it to be treated as a quality-of-implementation issue.

FUTURE DIRECTIONS

None.

SEE ALSO

sync( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

Aligned with *fsync*( ) in the POSIX Realtime Extension. Specifically, the DESCRIPTION and RETURN VALUE sections are much expanded, and the ERRORS section is updated to indicate that *fsync*( ) can return the error conditions defined for *read*( ) and *write*( ).

Issue 6

This function is marked as part of the File Synchronization option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EINVAL] and [EIO] mandatory error conditions are added.

14869 **NAME**

14870 ftell, ftello — return a file offset in a stream

14871 **SYNOPSIS**

14872 `#include <stdio.h>`

14873 `long ftell(FILE *stream);`

14874 CX `off_t ftello(FILE *stream);`

14875

14876 **DESCRIPTION**

14877 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14878 conflict between the requirements described here and the ISO C standard is unintentional. This
14879 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

14880 The *ftell*( ) function shall obtain the current value of the file-position indicator for the stream
14881 pointed to by *stream*.

14882 CX The *ftello*( ) function shall be equivalent to *ftell*( ), except that the return value is of type **off_t**.

14883 **RETURN VALUE**

14884 CX Upon successful completion, *ftell*( ) and *ftello*( ) shall return the current value of the file-position
14885 indicator for the stream measured in bytes from the beginning of the file.

14886 CX Otherwise, *ftell*( ) and *ftello*( ) shall return –1, cast to **long** and **off_t** respectively, and set *errno* to
14887 indicate the error.

14888 **ERRORS**

14889 CX The *ftell*( ) and *ftello*( ) functions shall fail if:

14890 CX [EBADF] The file descriptor underlying *stream* is not an open file descriptor.

14891 CX [EOVERFLOW] For *ftell*( ), the current file offset cannot be represented correctly in an object of
14892 type **long**.

14893 CX [EOVERFLOW] For *ftello*( ), the current file offset cannot be represented correctly in an object
14894 of type **off_t**.

14895 CX [ESPIPE] The file descriptor underlying *stream* is associated with a pipe or FIFO.

14896 The *ftell*( ) function may fail if:

14897 CX [ESPIPE] The file descriptor underlying *stream* is associated with a socket.

14898 **EXAMPLES**

14899 None.

14900 **APPLICATION USAGE**

14901 None.

14902 **RATIONALE**

14903 None.

14904 **FUTURE DIRECTIONS**

14905 None.

14906 **SEE ALSO**

14907 *fgetpos*( ), *fopen*( ), *fseek*( ), *lseek*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

14908 **CHANGE HISTORY**

14909 First released in Issue 1. Derived from Issue 1 of the SVID.

14910 **Issue 5**

14911 Large File Summit extensions are added.

14912 **Issue 6**

14913 Extensions beyond the ISO C standard are now marked.

14914 The following new requirements on POSIX implementations derive from alignment with the
14915 Single UNIX Specification:

14916 - The *ftello*( ) function is added.

14917 - The [EOVERFLOW] error conditions are added.

14918 An additional [ESPIPE] error condition is added for sockets.

14919 **NAME**

14920 ftime — get date and time (**LEGACY**)

14921 **SYNOPSIS**

14922 XSI
```
#include <sys/timeb.h>
```
14923
```
int ftime(struct timeb *tp);
```
14924

14925 **DESCRIPTION**

14926 The *ftime*( ) function shall set the *time* and *millitm* members of the **timeb** structure pointed to by
14927 *tp* to contain the seconds and milliseconds portions, respectively, of the current time in seconds
14928 since the Epoch. The contents of the *timezone* and *dstflag* members of *tp* after a call to *ftime*( ) are
14929 unspecified.

14930 The system clock need not have millisecond granularity. Depending on any granularity
14931 (particularly a granularity of one) renders code non-portable.

14932 **RETURN VALUE**

14933 Upon successful completion, the *ftime*( ) function shall return 0; otherwise, –1 shall be returned.

14934 **ERRORS**

14935 No errors are defined.

14936 **EXAMPLES**

14937 **Getting the Current Time and Date**

14938 The following example shows how to get the current system time values using the *ftime*( )
14939 function. The **timeb** structure pointed to by *tp* is filled with the current system time values for
14940 *time* and *millitm*.

14941
```
#include <sys/timeb.h>
```
14942
```
struct timeb tp;
```
14943
```
int          status;
```
14944
```
...
```
14945
```
status = ftime(&tp);
```

14946 **APPLICATION USAGE**

14947 For applications portability, the *time*( ) function should be used to determine the current time
14948 instead of *ftime*( ). Realtime applications should use *clock_gettime*( ) to determine the current
14949 time instead of *ftime*( ).

14950 **RATIONALE**

14951 None.

14952 **FUTURE DIRECTIONS**

14953 This function may be withdrawn in a future version.

14954 **SEE ALSO**

14955 *clock_getres*( ), *ctime*( ), *gettimeofday*( ), *time*( ), the Base Definitions volume of
14956 IEEE Std 1003.1-200x, **<sys/timeb.h>**

14957 **CHANGE HISTORY**

14958 First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

Issue 6

This function is marked LEGACY.

The DESCRIPTION is updated to refer to ‘‘seconds since the Epoch’’ rather than ‘‘seconds since 00:00:00 UTC (Coordinated Universal Time), January 1 1970’’ for consistency with other *time* functions.

14968 **NAME**

14969 ftok — generate an IPC key

14970 **SYNOPSIS**

14971 XSI
```
#include <sys/ipc.h>
```

14972
```
key_t ftok(const char *path, int id);
```

14973

14974 **DESCRIPTION**

14975 The *ftok*( ) function shall return a key based on *path* and *id* that is usable in subsequent calls to
14976 *msgget*( ), *semget*( ), and *shmget*( ). The application shall ensure that the *path* argument is the |
14977 pathname of an existing file that the process is able to *stat*( ). |

14978 The *ftok*( ) function shall return the same key value for all paths that name the same file, when
14979 called with the same *id* value, and return different key values when called with different *id*
14980 values or with paths that name different files existing on the same file system at the same time. It
14981 is unspecified whether *ftok*( ) shall return the same key value when called again after the file
14982 named by *path* is removed and recreated with the same name.

14983 Only the low order 8-bits of *id* are significant. The behavior of *ftok*( ) is unspecified if these bits
14984 are 0.

14985 **RETURN VALUE**

14986 Upon successful completion, *ftok*( ) shall return a key. Otherwise, *ftok*( ) shall return (**key_t**)–1
14987 and set *errno* to indicate the error.

14988 **ERRORS**

14989 The *ftok*( ) function shall fail if:

14990 [EACCES] Search permission is denied for a component of the path prefix.

14991 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
14992 argument.

14993 [ENAMETOOLONG]
14994 The length of the *path* argument exceeds {PATH_MAX} or a pathname |
14995 component is longer than {NAME_MAX}. |

14996 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

14997 [ENOTDIR] A component of the path prefix is not a directory.

14998 The *ftok*( ) function may fail if:

14999 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
15000 resolution of the *path* argument.

15001 [ENAMETOOLONG]
15002 Pathname resolution of a symbolic link produced an intermediate result |
15003 whose length exceeds {PATH_MAX}.

15004 **EXAMPLES**

15005 **Getting an IPC Key**

15006 The following example gets a unique key that can be used by the IPC functions *semget*( ),
15007 *msgget*( ), and *shmget*( ). The key returned by *ftok*( ) for this example is based on the ID value *S* |
15008 and the pathname **/tmp**. |

```
15009 #include <sys/ipc.h>
15010 ...
15011 key_t key;
15012 char *path = "/tmp";
15013 int id = 'S';
15014 key = ftok(path, id);
```

15015 **Saving an IPC Key**

15016 The following example gets a unique key based on the pathname **/tmp** and the ID value *a*. It |
15017 also assigns the value of the resulting key to the *semkey* variable so that it will be available to a
15018 later call to *semget*( ), *msgget*( ), or *shmget*( ).

```
15019 #include <sys/ipc.h>
15020 ...
15021 key_t semkey;
15022 if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
15023     perror("IPC error: ftok"); exit(1);
15024 }
```

15025 **APPLICATION USAGE**

15026 For maximum portability, *id* should be a single-byte character.

15027 **RATIONALE**

15028 None.

15029 **FUTURE DIRECTIONS**

15030 None.

15031 **SEE ALSO**

15032 *msgget*( ), *semget*( ), *shmget*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/ipc.h>**

15033 **CHANGE HISTORY**

15034 First released in Issue 4, Version 2.

15035 **Issue 5**

15036 Moved from X/OPEN UNIX extension to BASE.

15037 **Issue 6**

15038 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

15039 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
15040 [ELOOP] error condition is added.

15041 **NAME**

15042 ftruncate — truncate a file to a specified length

15043 **SYNOPSIS**

15044 `#include <unistd.h>`

15045 `int ftruncate(int fildes, off_t length);`

15046 **DESCRIPTION**

15047 If *fildes* is not a valid file descriptor open for writing, the *ftruncate*( ) function shall fail.

15048 If *fildes* refers to a regular file, the *ftruncate*( ) function shall cause the size of the file to be
15049 truncated to *length*. If the size of the file previously exceeded *length*, the extra data shall no
15050 longer be available to reads on the file. If the file previously was smaller than this size,
15051 XSI *ftruncate*( ) shall either increase the size of the file or fail. XSI-conformant systems shall increase
15052 the size of the file. If the file size is increased, the extended area shall appear as if it were zero-
15053 filled. The value of the seek pointer shall not be modified by a call to *ftruncate*( ).

15054 Upon successful completion, if *fildes* refers to a regular file, the *ftruncate*( ) function shall mark
15055 for update the *st_ctime* and *st_mtime* fields of the file and the S_ISUID and S_ISGID bits of the file
15056 mode may be cleared. If the *ftruncate*( ) function is unsuccessful, the file is unaffected.

15057 XSI If the request would cause the file size to exceed the soft file size limit for the process, the
15058 request shall fail and the implementation shall generate the SIGXFSZ signal for the thread.

15059 If *fildes* refers to a directory, *ftruncate*( ) shall fail.

15060 If *fildes* refers to any other file type, except a shared memory object, the result is unspecified.

15061 SHM If *fildes* refers to a shared memory object, *ftruncate*( ) shall set the size of the shared memory
15062 object to *length*.

15063 MF|SHM If the effect of *ftruncate*( ) is to decrease the size of a shared memory object or memory mapped
15064 file and whole pages beyond the new end were previously mapped, then the whole pages
15065 beyond the new end shall be discarded.

15066 MPR If the Memory Protection option is supported, references to discarded pages shall result in the
15067 generation of a SIGBUS signal; otherwise, the result of such references is undefined.

15068 MF|SHM If the effect of *ftruncate*( ) is to increase the size of a shared memory object, it is unspecified if the
15069 contents of any mapped pages between the old end-of-file and the new are flushed to the
15070 underlying object.

15071 **RETURN VALUE**

15072 Upon successful completion, *ftruncate*( ) shall return 0; otherwise, –1 shall be returned and *errno*
15073 set to indicate the error.

15074 **ERRORS**

15075 The *ftruncate*( ) function shall fail if:

15076 [EINTR] A signal was caught during execution.

15077 [EINVAL] The *length* argument was less than 0.

15078 [EFBIG] or [EINVAL]
15079 The *length* argument was greater than the maximum file size.

15080 XSI [EFBIG] The file is a regular file and *length* is greater than the offset maximum
15081 established in the open file description associated with *fildes*.

15082 [EIO] An I/O error occurred while reading from or writing to a file system.

[EBADF] or [EINVAL]
The *fildes* argument is not a file descriptor open for writing.

[EINVAL] The *fildes* argument references a file that was opened without write permission.

[EROFS] The named file resides on a read-only file system.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *ftruncate*( ) function is part of IEEE Std 1003.1-200x as it was deemed to be more useful than *truncate*( ). The *truncate*( ) function is provided as an XSI extension.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), *truncate*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE and aligned with *ftruncate*( ) in the POSIX Realtime Extension. Specifically, the DESCRIPTION is extensively reworded and [EROFS] is added to the list of mandatory errors that can be returned by *ftruncate*( ).

Large File Summit extensions are added.

Issue 6

The *truncate*( ) function has been split out into a separate reference page.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is change to indicate that if the file size is changed, and if the file is a regular file, the S_ISUID and S_ISGID bits in the file mode may be cleared.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The DESCRIPTION text is updated.

XSI-conformant systems are required to increase the size of the file if the file was previously smaller than the size requested.

15116 **NAME**

15117 ftrylockfile — stdio locking functions

15118 **SYNOPSIS**

15119 TSF
```
#include <stdio.h>
```

15120
```
int ftrylockfile(FILE *file);
```

15121

15122 **DESCRIPTION**

15123 Refer to *flockfile*( ).

15124 **NAME**

15125 ftw — traverse (walk) a file tree

15126 **SYNOPSIS**

15127 XSI
```
#include <ftw.h>

int ftw(const char *path, int (*fn)(const char *,
    const struct stat *ptr, int flag), int ndirs);
```
15128
15129
15130

15131 **DESCRIPTION**

15132 The *ftw*( ) function shall recursively descend the directory hierarchy rooted in *path*. For each
15133 object in the hierarchy, *ftw*( ) shall call the function pointed to by *fn*, passing it a pointer to a
15134 null-terminated character string containing the name of the object, a pointer to a **stat** structure
15135 containing information about the object, and an integer. Possible values of the integer, defined
15136 in the **<ftw.h>** header, are:

15137 FTW_D For a directory.

15138 FTW_DNR For a directory that cannot be read.

15139 FTW_F For a file.

15140 FTW_SL For a symbolic link (but see also FTW_NS below).

15141 FTW_NS For an object other than a symbolic link on which *stat*( ) could not successfully be
15142 executed. If the object is a symbolic link and *stat*( ) failed, it is unspecified whether
15143 *ftw*( ) passes FTW_SL or FTW_NS to the user-supplied function.

15144 If the integer is FTW_DNR, descendants of that directory shall not be processed. If the integer is
15145 FTW_NS, the **stat** structure contains undefined values. An example of an object that would
15146 cause FTW_NS to be passed to the function pointed to by *fn* would be a file in a directory with
15147 read but without execute (search) permission.

15148 The *ftw*( ) function shall visit a directory before visiting any of its descendants.

15149 The *ftw*( ) function shall use at most one file descriptor for each level in the tree.

15150 The argument *ndirs* should be in the range of 1 to {OPEN_MAX}.

15151 The tree traversal shall continue until either the tree is exhausted, an invocation of *fn* returns a
15152 non-zero value, or some error, other than [EACCES], is detected within *ftw*( ).

15153 The *ndirs* argument shall specify the maximum number of directory streams or file descriptors
15154 or both available for use by *ftw*( ) while traversing the tree. When *ftw*( ) returns it shall close any
15155 directory streams and file descriptors it uses not counting any opened by the application-
15156 supplied *fn* function.

15157 The results are unspecified if the application-supplied *fn* function does not preserve the current
15158 working directory.

15159 The *ftw*( ) function need not be reentrant. A function that is not required to be reentrant is not
15160 required to be thread-safe.

15161 **RETURN VALUE**

15162 If the tree is exhausted, *ftw*( ) shall return 0. If the function pointed to by *fn* returns a non-zero
15163 value, *ftw*( ) shall stop its tree traversal and return whatever value was returned by the function
15164 pointed to by *fn*. If *ftw*( ) detects an error, it shall return –1 and set *errno* to indicate the error.

15165 If *ftw*( ) encounters an error other than [EACCES] (see FTW_DNR and FTW_NS above), it shall
15166 return –1 and set *errno* to indicate the error. The external variable *errno* may contain any error

value that is possible when a directory is opened or when one of the *stat* functions is executed on a directory or file.

ERRORS

The *ftw*() function shall fail if:

[EACCES] Search permission is denied for any component of *path* or read permission is denied for *path*.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}. |

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

[ENOTDIR] A component of *path* is not a directory. |

[EOVERFLOW] A field in the **stat** structure cannot be represented correctly in the current programming environment for one or more files found in the file hierarchy. | |

The *ftw*() function may fail if:

[EINVAL] The value of the *ndirs* argument is invalid.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}. |

In addition, if the function pointed to by *fn* encounters system errors, *errno* may be set accordingly.

EXAMPLES

Walking a Directory Structure

The following example walks the current directory structure, calling the *fn* function for every directory entry, using at most 10 file descriptors:

```
#include <ftw.h>
...
if (ftw(".", fn, 10) != 0) {
    perror("ftw"); exit(2);
}
```

APPLICATION USAGE

The *ftw*() function may allocate dynamic storage during its operation. If *ftw*() is forcibly terminated, such as by *longjmp*() or *siglongjmp*() being executed by the function pointed to by *fn* or an interrupt routine, *ftw*() does not have a chance to free that storage, so it remains permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function pointed to by *fn* return a non-zero value at its next invocation.

15207 **RATIONALE**

15208 None.

15209 **FUTURE DIRECTIONS**

15210 None.

15211 **SEE ALSO**

15212 *longjmp*( ), *lstat*( ), *malloc*( ), *nftw*( ), *opendir*( ), *siglongjmp*( ), *stat*( ), the Base Definitions volume of
15213 IEEE Std 1003.1-200x, **<ftw.h>**, **<sys/stat.h>**

15214 **CHANGE HISTORY**

15215 First released in Issue 1. Derived from Issue 1 of the SVID.

15216 **Issue 5**

15217 UX codings in the DESCRIPTION, RETURN VALUE, and ERRORS sections have been changed
15218 to EX.

15219 **Issue 6**

15220 The ERRORS section is updated as follows: |

15221 • The wording of the mandatory [ELOOP] error condition is updated. |

15222 • A second optional [ELOOP] error condition is added. |

15223 • The [EOVERFLOW] mandatory error condition is added. |

15224 Text is added to the DESCRIPTION to say that the *ftw*( ) function need not be reentrant and that |
15225 the results are unspecified if the application-supplied *fn* function does not preserve the current |
15226 working directory. |

15227 **NAME**

15228 funlockfile — stdio locking functions

15229 **SYNOPSIS**

15230 TSF
```
#include <stdio.h>
```

15231
```
void funlockfile(FILE *file);
```

15232

15233 **DESCRIPTION**

15234 Refer to *flockfile*( ).

15235 **NAME**

15236 fwide — set stream orientation

15237 **SYNOPSIS**

15238 ```
#include <stdio.h>
```

15239 ```
#include <wchar.h>
```

15240 ```
int fwide(FILE *stream, int mode);
```

15241 **DESCRIPTION**

15242 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15243 conflict between the requirements described here and the ISO C standard is unintentional. This
15244 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

15245 The *fwide*( ) function shall determine the orientation of the stream pointed to by *stream*. If *mode* is
15246 greater than zero, the function first attempts to make the stream wide-oriented. If *mode* is less
15247 than zero, the function first attempts to make the stream byte-oriented. Otherwise, *mode* is zero
15248 and the function does not alter the orientation of the stream.

15249 If the orientation of the stream has already been determined, *fwide*( ) shall not change it.

15250 CX Since no return value is reserved to indicate an error, an application wishing to check for error |
15251 situations should set *errno* to 0, then call *fwide*( ), then check *errno*, and if it is non-zero, assume
15252 an error has occurred.

15253 **RETURN VALUE**

15254 The *fwide*( ) function shall return a value greater than zero if, after the call, the stream has wide-
15255 orientation, a value less than zero if the stream has byte-orientation, or zero if the stream has no
15256 orientation.

15257 **ERRORS**

15258 The *fwide*( ) function may fail if:

15259 CX [EBADF] The *stream* argument is not a valid stream.

15260 **EXAMPLES**

15261 None.

15262 **APPLICATION USAGE**

15263 A call to *fwide*( ) with *mode* set to zero can be used to determine the current orientation of a
15264 stream.

15265 **RATIONALE**

15266 None.

15267 **FUTURE DIRECTIONS**

15268 None.

15269 **SEE ALSO**

15270 The Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

15271 **CHANGE HISTORY**

15272 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
15273 (E).

15274 **Issue 6**

15275 Extensions beyond the ISO C standard are now marked.
```

15276 **NAME**

15277 fwprintf, swprintf, wprintf — print formatted wide-character output

15278 **SYNOPSIS**

15279 ```
#include <stdio.h>
```
15280 ```
#include <wchar.h>
```

15281 ```
int fwprintf(FILE *restrict stream, const wchar_t *restrict format, ...);
```
15282 ```
int swprintf(wchar_t *restrict ws, size_t n,
```
15283 ```
    const wchar_t *restrict format, ...);
```
15284 ```
int wprintf(const wchar_t *restrict format, ...);
```

15285 **DESCRIPTION**

15286 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15287 conflict between the requirements described here and the ISO C standard is unintentional. This
15288 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

15289 The *fwprintf*() function shall place output on the named output *stream*. The *wprintf*() function |
15290 shall place output on the standard output stream *stdout*. The *swprintf*() function shall place |
15291 output followed by the null wide character in consecutive wide characters starting at *ws*; no |
15292 more than *n* wide characters shall be written, including a terminating null wide character, which |
15293 is always added (unless *n* is zero).

15294 Each of these functions shall convert, format, and print its arguments under control of the *format* |
15295 wide-character string. The *format* is composed of zero or more directives: *ordinary wide-*
15296 *characters*, which are simply copied to the output stream, and *conversion specifications*, each of
15297 which results in the fetching of zero or more arguments. The results are undefined if there are
15298 insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the
15299 excess arguments are evaluated but are otherwise ignored.

15300 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
15301 to the next unused argument. In this case, the conversion specifier wide character % (see below)
15302 is replaced by the sequence "%*n*$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}],
15303 giving the position of the argument in the argument list. This feature provides for the definition
15304 of *format* wide-character strings that select arguments in an order appropriate to specific
15305 languages (see the EXAMPLES section).

15306 The *format* can contain either numbered argument specifications (that is, "%*n*$" and "**m*$"), or
15307 unnumbered argument conversion specifications (that is, % and *), but not both. The only |
15308 exception to this is that %% can be mixed with the "%*n*$" form. The results of mixing numbered |
15309 and unnumbered argument specifications in a *format* wide-character string are undefined. When
15310 numbered argument specifications are used, specifying the *N*th argument requires that all the
15311 leading arguments, from the first to the (*N*–1)th, are specified in the format wide-character
15312 string.

15313 In *format* wide-character strings containing the "%*n*$" form of conversion specification,
15314 numbered arguments in the argument list can be referenced from the *format* wide-character
15315 string as many times as required.

15316 In *format* wide-character strings containing the % form of conversion specification, each |
15317 argument in the argument list shall be used exactly once. |

15318 CX All forms of the *fwprintf*() function allow for the insertion of a locale-dependent radix character
15319 in the output string, output as a wide-character value. The radix character is defined in the
15320 program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix
15321 character is not defined, the radix character shall default to a period ('.').

15322 XSI Each conversion specification is introduced by the '%' wide character or by the wide-character
15323 sequence "%*n*$", after which the following appear in sequence:

15324 • Zero or more *flags* (in any order), which modify the meaning of the conversion specification.

15325 • An optional minimum *field width*. If the converted value has fewer wide characters than the
15326 field width, it shall be padded with spaces by default on the left; it shall be padded on the
15327 right, if the left-adjustment flag ('–'), described below, is given to the field width. The field
15328 width takes the form of an asterisk ('*'), described below, or a decimal integer.

15329 • An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u, x,
15330 and X conversion specifiers; the number of digits to appear after the radix character for the a,
15331 A, e, E, f, and F conversion specifiers; the maximum number of significant digits for the g
15332 and G conversion specifiers; or the maximum number of wide characters to be printed from a
15333 string in the s conversion specifiers. The precision takes the form of a period ('.') followed
15334 either by an asterisk ('*'), described below, or an optional decimal digit string, where a null
15335 digit string is treated as 0. If a precision appears with any other conversion wide character,
15336 the behavior is undefined.

15337 • An optional length modifier that specifies the size of the argument.

15338 • A *conversion specifier* wide character that indicates the type of conversion to be applied.

15339 A field width, or precision, or both, may be indicated by an asterisk ('*'). In this case an
15340 argument of type **int** supplies the field width or precision. Applications shall ensure that
15341 arguments specifying field width, or precision, or both appear in that order before the argument,
15342 if any, to be converted. A negative field width is taken as a '–' flag followed by a positive field
15343 XSI width. A negative precision is taken as if the precision were omitted. In format wide-character
15344 strings containing the "%*n*$" form of a conversion specification, a field width or precision may
15345 be indicated by the sequence "**m*$", where *m* is a decimal integer in the range
15346 [1,{NL_ARGMAX}] giving the position in the argument list (after the format argument) of an
15347 integer argument containing the field width or precision, for example:

15348 ```
wprintf(L"%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

15349 The flag wide characters and their meanings are:

15350 XSI ' The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %F, %g, or %G)
15351 shall be formatted with thousands' grouping wide characters. For other conversions,
15352 the behavior is undefined. The numeric grouping wide character is used.

15353 – The result of the conversion shall be left-justified within the field. The conversion shall
15354 be right-justified if this flag is not specified.

15355 + The result of a signed conversion shall always begin with a sign ('+' or '–'). The
15356 conversion shall begin with a sign only when a negative value is converted if this flag is
15357 not specified.

15358 <space> If the first wide character of a signed conversion is not a sign, or if a signed conversion
15359 results in no wide characters, a <space> shall be prefixed to the result. This means that
15360 if the <space> and '+' flags both appear, the <space> flag shall be ignored.

15361 # Specifies that the value is to be converted to an alternative form. For o conversion, it |
15362 increases the precision (if necessary) to force the first digit of the result to be 0. For x or
15363 X conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For a, A, e,
15364 E, f, F, g, and G conversion specifiers, the result shall always contain a radix character,
15365 even if no digits follow it. Without this flag, a radix character appears in the result of
15366 these conversions only if a digit follows it. For g and G conversion specifiers, trailing
15367 zeros shall *not* be removed from the result as they normally are. For other conversion
```

specifiers, the behavior is undefined.

`0` For `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the `'0'` and `'-'` flags both appear, the `'0'` flag shall be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversion specifiers, if a precision is specified, the `'0'` flag shall be ignored. If the `'0'` and `'\''` flags both appear, the grouping wide characters are inserted before zero padding. For other conversions, the behavior is undefined.

The length modifiers and their meanings are:

`hh` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **signed char** or **unsigned char** before printing); or that a following `n` conversion specifier applies to a pointer to a **signed char** argument.

`h` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **short** or **unsigned short** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **short** or **unsigned short** before printing); or that a following `n` conversion specifier applies to a pointer to a **short** argument.

`l` (ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **long** or **unsigned long** argument; that a following `n` conversion specifier applies to a pointer to a **long** argument; that a following `c` conversion specifier applies to a **wint_t** argument; that a following `s` conversion specifier applies to a pointer to a **wchar_t** argument; or has no effect on a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.

`ll` (ell-ell)
Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **long long** or **unsigned long long** argument; or that a following `n` conversion specifier applies to a pointer to a **long long** argument.

`j` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an **intmax_t** or **uintmax_t** argument; or that a following `n` conversion specifier applies to a pointer to an **intmax_t** argument.

`z` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **size_t** or the corresponding signed integer type argument; or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to **size_t** argument.

`t` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **ptrdiff_t** or the corresponding **unsigned** type argument; or that a following `n` conversion specifier applies to a pointer to a **ptrdiff_t** argument.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a **long double** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

`d`, `i` The **int** argument shall be converted to a signed decimal in the style `"[-]`*dddd*`"`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision

of zero shall be no wide characters.

`o` The **unsigned** argument shall be converted to unsigned octal format in the style `"dddd"`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision of zero shall be no wide characters.

`u` The **unsigned** argument shall be converted to unsigned decimal format in the style `"dddd"`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision of zero shall be no wide characters.

`x` The **unsigned** argument shall be converted to unsigned hexadecimal format in the style `"dddd"`; the letters `"abcdef"` are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision of zero shall be no wide characters.

`X` Equivalent to the `x` conversion specifier, except that letters `"ABCDEF"` are used instead of `"abcdef"`.

`f`, `F` The **double** argument shall be converted to decimal notation in the style `"[-]`*`ddd.ddd`*`"`, where the number of digits after the radix character shall be equal to the precision specification. If the precision is missing, it shall be taken as 6; if the precision is explicitly zero and no `'#'` flag is present, no radix character shall appear. If a radix character appears, at least one digit shall appear before it. The value shall be rounded in an implementation-defined manner to the appropriate number of digits.

A **double** argument representing an infinity shall be converted in one of the styles `"[-]inf"` or `"[-]infinity"`; which style is implementation-defined. A **double** argument representing a NaN shall be converted in one of the styles `"[-]nan"` or `"[-]nan(`*`n-char-sequence`*`)"`; which style, and the meaning of any *n-char-sequence*, is implementation-defined. The `F` conversion specifier produces `"INF"`, `"INFINITY"`, or `"NAN"` instead of `"inf"`, `"infinity"`, or `"nan"`, respectively.

e, *E* The **double** argument shall be converted in the style `"[-]`*`d.ddd`*`e±dd"`, where there shall be one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it shall be equal to the precision; if the precision is missing, it shall be taken as 6; if the precision is zero and no `'#'` flag is present, no radix character shall appear. The value shall be rounded in an implementation-defined manner to the appropriate number of digits. The `E` conversion wide character shall produce a number with `'E'` instead of `'e'` introducing the exponent. The exponent always shall contain at least two digits. If the value is zero, the exponent shall be zero.

A **double** argument representing an infinity or NaN shall be converted in the style of an `f` or `F` conversion specifier.

`g`, `G` The **double** argument shall be converted in the style `f` or `e` (or in the style `F` or `E` in the case of a `G` conversion specifier), with the precision specifying the number of significant digits. If an explicit precision is zero, it shall be taken as 1. The style used depends on the value converted; style `e` (or `E`) shall be used only if the exponent resulting from such a conversion is less than –4 or greater than or equal to the precision. Trailing zeros shall be removed from the fractional portion of the result; a radix character shall appear only if it is followed by a digit.

A **double** argument representing an infinity or NaN shall be converted in the style of an `f` or `F` conversion specifier.

`a`, `A` A **double** argument representing a floating-point number shlal be converted in the style `"[-]0xh.hhhhp±d"`, where there shall be one hexadecimal digit (which is non-zero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point wide character and the number of hexadecimal digits after it shall be equal to the precision; if the precision is missing and FLT_RADIX is a power of 2, then the precision shall be sufficient for an exact representation of the value; if the precision is missing and FLT_RADIX is not a power of 2, then the precision shall be sufficient to distinguish values of type **double**, except that trailing zeros may be omitted; if the precision is zero and the `'#'` flag is not specified, no decimal-point wide character shall appear. The letters `"abcdef"` are used for `a` conversion and the letters `"ABCDEF"` for `A` conversion. The `A` conversion specifier produces a number with `'X'` and `'P'` instead of `'x'` and `'p'`. The exponent shall always contain at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent shall be zero.

A **double** argument representing an infinity or NaN shall be converted in the style of an `f` or `F` conversion specifier.

`c` If no `l` (ell) qualifier is present, the **int** argument shall be converted to a wide character as if by calling the *btowc*( ) function and the resulting wide character shall be written. Otherwise, the **wint_t** argument shall be converted to **wchar_t**, and written.

`s` If no `l` (ell) qualifier is present, the application shall ensure that the argument is a pointer to a character array containing a character sequence beginning in the initial shift state. Characters from the array shall be converted as if by repeated calls to the *mbrtowc*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first character is converted, and written up to (but not including) the terminating null wide character. If the precision is specified, no more than that many wide characters shall be written. If the precision is not specified, or is greater than the size of the array, the application shall ensure that the array contains a null wide character.

If an `l` (ell) qualifier is present, the application shall ensure that the argument is a pointer to an array of type **wchar_t**. Wide characters from the array shall be written up to (but not including) a terminating null wide character. If no precision is specified, or is greater than the size of the array, the application shall ensure that the array contains a null wide character. If a precision is specified, no more than that many wide characters shall be written.

`p` The application shall ensure that the argument is a pointer to **void**. The value of the pointer shall be converted to a sequence of printable wide characters in an implementation-defined manner.

`n` The application shall ensure that the argument is a pointer to an integer into which is written the number of wide characters written to the output so far by this call to one of the *fwprintf*( ) functions. No argument shall be converted, but one shall be consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

XSI `C` Equivalent to `lc`.

XSI `S` Equivalent to `ls`.

15507 % Output a '%' wide character; no argument shall be converted. The entire conversion |
15508 specification shall be %%.

15509 If a conversion specification does not match one of the above forms, the behavior is undefined.

15510 In no case does a nonexistent or small field width cause truncation of a field; if the result of a
15511 conversion is wider than the field width, the field shall be expanded to contain the conversion |
15512 result. Characters generated by *fwprintf*( ) and *wprintf*( ) shall be printed as if *fputwc*( ) had been |
15513 called.

15514 For a and A conversions, if FLT_RADIX is not a power of 2 and the result is not exactly |
15515 representable in the given precision, the result should be one of the two adjacent numbers in |
15516 hexadecimal floating style with the given precision, with the extra stipulation that the error |
15517 should have a correct sign for the current rounding direction. |

15518 For e, E, f, F, g, and G conversion specifiers, if the number of significant decimal digits is at most |
15519 DECIMAL_DIG, then the result should be correctly rounded. If the number of significant
15520 decimal digits is more than DECIMAL_DIG but the source value is exactly representable with
15521 DECIMAL_DIG digits, then the result should be an exact representation with trailing zeros.
15522 Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having
15523 DECIMAL_DIG significant digits; the value of the resultant decimal string D should satisfy $L <= D <= U$, with the extra stipulation that the error should have a correct sign for the current
15524 rounding direction.
15525

15526 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the call to a
15527 successful execution of *fwprintf*( ) or *wprintf*( ) and the next successful completion of a call to
15528 *fflush*( ) or *fclose*( ) on the same stream, or a call to *exit*( ) or *abort*( ).

15529 **RETURN VALUE**

15530 Upon successful completion, these functions shall return the number of wide characters
15531 transmitted, excluding the terminating null wide character in the case of *swprintf*( ), or a negative
15532 CX value if an output error was encountered, and set *errno* to indicate the error.

15533 If *n* or more wide characters were requested to be written, *swprintf*( ) shall return a negative
15534 CX value, and set *errno* to indicate the error.

15535 **ERRORS**

15536 For the conditions under which *fwprintf*( ) and *wprintf*( ) fail and may fail, refer to *fputwc*( ).

15537 In addition, all forms of *fwprintf*( ) may fail if:

15538 XSI [EILSEQ] A wide-character code that does not correspond to a valid character has been
15539 detected.

15540 XSI [EINVAL] There are insufficient arguments.

15541 In addition, *wprintf*( ) and *fwprintf*( ) may fail if:

15542 XSI [ENOMEM] Insufficient storage space is available.

EXAMPLES

To print the language-independent date and time format, the following statement could be used:

```
wprintf(format, weekday, month, day, hour, min);
```

For American usage, *format* could be a pointer to the wide-character string:

```
L"%s, %s %d, %d:%.2d\n"
```

producing the message:

```
Sunday, July 3, 10:02
```

whereas for German usage, *format* could be a pointer to the wide-character string:

```
L"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

producing the message:

```
Sonntag, 3. Juli, 10:02
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

btowc( ), *fputwc*( ), *fwscanf*( ), *mbrtowc*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The Open Group Corrigendum U040/1 is applied to the RETURN VALUE section, describing the case if *n* or more wide characters are requested to be written using *swprintf*( ).

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The prototypes for *fwprintf*( ), *swprintf*( ), and *wprintf*( ) are updated.
- The DESCRIPTION is updated.
- The `hh`, `ll`, `j`, `t`, and `z` length modifiers are added.
- The `a`, `A`, and `F` conversion characters are added.
- XSI shading is removed from the description of character string representations of infinity and NaN floating-point values.

The DESCRIPTION is updated to use the terms ‘‘conversion specifier’’ and ‘‘conversion specification’’ consistently.

ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

15581 **NAME**

15582 fwrite — binary output

15583 **SYNOPSIS**

15584
```
#include <stdio.h>
```

15585
```
size_t fwrite(const void *restrict ptr, size_t size, size_t nitems,
```
15586
```
    FILE *restrict stream);
```

15587 **DESCRIPTION**

15588 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15589 conflict between the requirements described here and the ISO C standard is unintentional. This
15590 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

15591 The *fwrite*( ) function shall write, from the array pointed to by *ptr*, up to *nitems* elements whose
15592 size is specified by *size*, to the stream pointed to by *stream*. For each object, *size* calls shall be
15593 made to the *fputc*( ) function, taking the values (in order) from an array of **unsigned char** exactly
15594 overlaying the object. The file-position indicator for the stream (if defined) shall be advanced by
15595 the number of bytes successfully written. If an error occurs, the resulting value of the file-
15596 position indicator for the stream is unspecified.

15597 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
15598 execution of *fwrite*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the
15599 same stream, or a call to *exit*( ) or *abort*( ).

15600 **RETURN VALUE**

15601 The *fwrite*( ) function shall return the number of elements successfully written, which may be
15602 less than *nitems* if a write error is encountered. If *size* or *nitems* is 0, *fwrite*( ) shall return 0 and the
15603 state of the stream remains unchanged. Otherwise, if a write error occurs, the error indicator for
15604 CX the stream shall be set, and *errno* shall be set to indicate the error.

15605 **ERRORS**

15606 Refer to *fputc*( ).

15607 **EXAMPLES**

15608 None.

15609 **APPLICATION USAGE**

15610 Because of possible differences in element length and byte ordering, files written using *fwrite*( )
15611 are application-dependent, and possibly cannot be read using *fread*( ) by a different application
15612 or by the same application on a different processor.

15613 **RATIONALE**

15614 None.

15615 **FUTURE DIRECTIONS**

15616 None.

15617 **SEE ALSO**

15618 *ferror*( ), *fopen*( ), *printf*( ), *putc*( ), *puts*( ), *write*( ), the Base Definitions volume of
15619 IEEE Std 1003.1-200x, **<stdio.h>**

15620 **CHANGE HISTORY**

15621 First released in Issue 1. Derived from Issue 1 of the SVID.

15622 **Issue 6**

15623 Extensions beyond the ISO C standard are now marked.

15624 The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

- The *fwrite*( ) prototype is updated.
- The DESCRIPTION is updated to clarify how the data is written out using *fputc*( ).

15627 **NAME**

15628 fwscanf, swscanf, wscanf — convert formatted wide-character input

15629 **SYNOPSIS**

15630 ```
#include <stdio.h>
```
15631 ```
#include <wchar.h>
```

15632 ```
int fwscanf(FILE *restrict stream, const wchar_t *restrict format, ...);
```
15633 ```
int swscanf(const wchar_t *restrict ws,
```
15634 ```
    const wchar_t *restrict format, ...);
```
15635 ```
int wscanf(const wchar_t *restrict format, ... );
```

15636 **DESCRIPTION**

15637 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15638 conflict between the requirements described here and the ISO C standard is unintentional. This
15639 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

15640 The *fwscanf*( ) function shall read from the named input *stream*. The *wscanf*( ) function shall read |
15641 from the standard input stream *stdin*. The *swscanf*( ) function shall read from the wide-character |
15642 string *ws*. Each function reads wide characters, interprets them according to a format, and stores |
15643 the results in its arguments. Each expects, as arguments, a control wide-character string *format*
15644 described below, and a set of *pointer* arguments indicating where the converted input should be
15645 stored. The result is undefined if there are insufficient arguments for the format. If the format is
15646 exhausted while arguments remain, the excess arguments are evaluated but are otherwise
15647 ignored.

15648 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
15649 to the next unused argument. In this case, the conversion specifier wide character % (see below)
15650 is replaced by the sequence "%n$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}].
15651 This feature provides for the definition of format wide-character strings that select arguments in
15652 an order appropriate to specific languages. In format wide-character strings containing the
15653 "*%n*$" form of conversion specifications, it is unspecified whether numbered arguments in the
15654 argument list can be referenced from the format wide-character string more than once.

15655 The *format* can contain either form of a conversion specification—that is, % or "*%n*$"— but the
15656 two forms cannot normally be mixed within a single *format* wide-character string. The only
15657 exception to this is that %% or %* can be mixed with the "*%n*$" form. When numbered |
15658 argument specifications are used, specifying the *N*th argument requires that all the leading |
15659 arguments, from the first to the (*N*–1)th, are pointers. |

15660 CX The *fwscanf*( ) function in all its forms allows for detection of a language-dependent radix
15661 character in the input string, encoded as a wide-character value. The radix character is defined in
15662 the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the
15663 radix character is not defined, the radix character shall default to a period (`'.'`).

15664 The *format* is a wide-character string composed of zero or more directives. Each directive is
15665 composed of one of the following: one or more white-space wide characters (<space>s, <tab>s,
15666 <newline>s, <vertical-tab>s, or <form-feed>s); an ordinary wide character (neither `'%'` nor a
15667 white-space character); or a conversion specification. Each conversion specification is introduced
15668 XSI by a `'%'` or the sequence "*%n*$" after which the following appear in sequence:

15669 - An optional assignment-suppressing character `'*'`.
15670 - An optional non-zero decimal integer that specifies the maximum field width.
15671 - An optional length modifier that specifies the size of the receiving object.

15672 - A conversion specifier wide character that specifies the type of conversion to be applied. The
15673 valid conversion specifiers are described below.

15674 The *fwscanf*( ) functions shall execut each directive of the format in turn. If a directive fails, as
15675 detailed below, the function shall return. Failures are described as input failures (due to the
15676 unavailability of input bytes) or matching failures (due to inappropriate input).

15677 A directive composed of one or more white-space wide characters is executed by reading input
15678 until no more valid input can be read, or up to the first wide character which is not a white-
15679 space wide character, which remains unread.

15680 A directive that is an ordinary wide character shall be executed as follows. The next wide
15681 character is read from the input and compared with the wide character that comprises the
15682 directive; if the comparison shows that they are not equivalent, the directive shall fail, and the
15683 differing and subsequent wide characters remain unread. Similarly, if end-of-file, an encoding
15684 error, or a read error prevents a wide character from being read, the directive shall fail.

15685 A directive that is a conversion specification defines a set of matching input sequences, as
15686 described below for each conversion wide character. A conversion specification is executed in
15687 the following steps.

15688 Input white-space wide characters (as specified by *iswspace*( )) shall be skipped, unless the
15689 conversion specification includes a [, c, or n conversion specifier.

15690 An item shall be read from the input, unless the conversion specification includes an n
15691 conversion specifier wide character. An input item is defined as the longest sequence of input
15692 wide characters, not exceeding any specified field width, which is an initial subsequence of a
15693 matching sequence. The first wide character, if any, after the input item shall remain unread. If
15694 the length of the input item is zero, the execution of the conversion specification shall fail; this
15695 condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented
15696 input from the stream, in which case it is an input failure.

15697 Except in the case of a % conversion specifier, the input item (or, in the case of a %n conversion
15698 specification, the count of input wide characters) shall be converted to a type appropriate to the
15699 conversion wide character. If the input item is not a matching sequence, the execution of the
15700 conversion specification shall fail; this condition is a matching failure. Unless assignment
15701 suppression was indicated by a '*', the result of the conversion shall be placed in the object
15702 pointed to by the first argument following the *format* argument that has not already received a
15703 XSI conversion result if the conversion specification is introduced by %, or in the *n*th argument if
15704 introduced by the wide-character sequence "%*n*$". If this object does not have an appropriate
15705 type, or if the result of the conversion cannot be represented in the space provided, the behavior
15706 is undefined.

15707 The length modifiers and their meanings are:

15708 hh Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
15709 argument with type pointer to **signed char** or **unsigned char**.

15710 h Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
15711 argument with type pointer to **short** or **unsigned short**.

15712 l (ell) Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
15713 argument with type pointer to **long** or **unsigned long**; that a following a, A, e, E, f, F, g,
15714 or G conversion specifier applies to an argument with type pointer to **double**; or that a
15715 following c, s, or [conversion specifier applies to an argument with type pointer to
15716 **wchar_t**.

15717 `ll` (ell-ell)
15718 Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an
15719 argument with type pointer to **long long** or **unsigned long long**.

15720 `j` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an
15721 argument with type pointer to **intmax_t** or **uintmax_t**.

15722 `z` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an
15723 argument with type pointer to **size_t** or the corresponding signed integer type.

15724 `t` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an
15725 argument with type pointer to **ptrdiff_t** or the corresponding **unsigned** type.

15726 `L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an
15727 argument with type pointer to **long double**.

15728 If a length modifier appears with any conversion specifier other than as specified above, the
15729 behavior is undefined.

15730 The following conversion specifier wide characters are valid:

15731 `d` Matches an optionally signed decimal integer, whose format is the same as expected for
15732 the subject sequence of *wcstol*( ) with the value 10 for the *base* argument. In the absence
15733 of a size modifier, the application shall ensure that the corresponding argument is a
15734 pointer to **int**.

15735 `i` Matches an optionally signed integer, whose format is the same as expected for the
15736 subject sequence of *wcstol*( ) with 0 for the *base* argument. In the absence of a size
15737 modifier, the application shall ensure that the corresponding argument is a pointer to
15738 **int**.

15739 `o` Matches an optionally signed octal integer, whose format is the same as expected for
15740 the subject sequence of *wcstoul*( ) with the value 8 for the *base* argument. In the absence
15741 of a size modifier, the application shall ensure that the corresponding argument is a
15742 pointer to **unsigned**.

15743 `u` Matches an optionally signed decimal integer, whose format is the same as expected for
15744 the subject sequence of *wcstoul*( ) with the value 10 for the *base* argument. In the absence
15745 of a size modifier, the application shall ensure that the corresponding argument is a
15746 pointer to **unsigned**.

15747 `x` Matches an optionally signed hexadecimal integer, whose format is the same as
15748 expected for the subject sequence of *wcstoul*( ) with the value 16 for the *base* argument.
15749 In the absence of a size modifier, the application shall ensure that the corresponding
15750 argument is a pointer to **unsigned**.

15751 `a`, `e`, `f`, `g`
15752 Matches an optionally signed floating-point number, infinity, or NaN whose format is
15753 the same as expected for the subject sequence of *wcstod*( ). In the absence of a size
15754 modifier, the application shall ensure that the corresponding argument is a pointer to
15755 **float**.

15756 If the *fwprintf*( ) family of functions generates character string representations for
15757 infinity and NaN (a symbolic entity encoded in floating-point format) to support
15758 IEEE Std 754-1985, the *fwscanf*( ) family of functions shall recognize them as input.

15759 `s` Matches a sequence of non white-space wide characters. If no `l` (ell) qualifier is present, |
15760 characters from the input field shall be converted as if by repeated calls to the |
15761 *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object

initialized to zero before the first wide character is converted. The application shall ensure that the corresponding argument is a pointer to a character array large enough to accept the sequence and the terminating null character, which shall be added automatically.

Otherwise, the application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically.

[Matches a non-empty sequence of wide characters from a set of expected wide characters (the *scanset*). If no `l` (ell) qualifier is present, wide characters from the input field shall be converted as if by repeated calls to the *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The application shall ensure that the corresponding argument is a pointer to a character array large enough to accept the sequence and the terminating null character, which shall be added automatically.

If an `l` (ell) qualifier is present, the application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically.

The conversion specification includes all subsequent wide characters in the *format* string up to and including the matching right square bracket (`']'`). The wide characters between the square brackets (the *scanlist*) comprise the scanset, unless the wide character after the left square bracket is a circumflex (`'^'`), in which case the scanset contains all wide characters that do not appear in the scanlist between the circumflex and the right square bracket. If the conversion specification begins with `"[]"` or `"[^]"`, the right square bracket is included in the scanlist and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise, the first right square bracket is the one that ends the conversion specification. If a `'-'` is in the scanlist and is not the first wide character, nor the second where the first wide character is a `'^'`, nor the last wide character, the behavior is implementation-defined.

c Matches a sequence of wide characters of exactly the number specified by the field width (1 if no field width is present in the conversion specification).

If no `l` (ell) length modifier is present, characters from the input field shall be converted as if by repeated calls to the *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

If an `l` (ell) length modifier is present, the corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence. No null wide character is added.

Otherwise, the application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the sequence. No null wide character is added.

p Matches an implementation-defined set of sequences, which shall be the same as the set of sequences that is produced by the `%p` conversion specification of the corresponding *fwprintf*( ) functions. The application shall ensure that the corresponding argument is a pointer to a pointer to **void**. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise, the

behavior of the %p conversion is undefined.

n — No input is consumed. The application shall ensure that the corresponding argument is a pointer to the integer into which is to be written the number of wide characters read from the input so far by this call to the *fwscanf*( ) functions. Execution of a %n conversion specification shall not increment the assignment count returned at the completion of execution of the function. No argument shall be converted, but one shall be consumed. If the conversion specification includes an assignment-suppressing wide character or a field width, the behavior is undefined.

XSI C — Equivalent to lc.

XSI S — Equivalent to ls.

% — Matches a single '%' wide character; no conversion or assignment shall occur. The complete conversion specification shall be %%.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers A, E, F, G, and X are also valid and shall be equivalent to, respectively, a, e, f, g, and x.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any wide characters matching the current conversion specification (except for %n) have been read (other than leading white-space, where permitted), execution of the current conversion specification shall terminate with an input failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if any) shall be terminated with an input failure.

Reaching the end of the string in *swscanf*( ) shall be equivalent to encountering end-of-file for *fwscanf*( ).

If conversion terminates on a conflicting input, the offending input shall be left unread in the input. Any trailing white space (including <newline>) shall be left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the %n conversion specification.

CX The *fwscanf*( ) and *wscanf*( ) functions may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgetwc*( ), *fgets*( ), *fgetws*( ), *fread*( ), *getc*( ), *getwc*( ), *getchar*( ), *getwchar*( ), *gets*( ), *fscanf*( ), or *fwscanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ).

RETURN VALUE

Upon successful completion, these functions shall return the number of successfully matched and assigned input items; this number can be zero in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF shall be returned. If a read CX error occurs the error indicator for the stream is set, EOF shall be returned, and *errno* shall be set to indicate the error.

ERRORS

For the conditions under which the *fwscanf*( ) functions shall fail and may fail, refer to *fgetwc*( ).

In addition, *fwscanf*( ) may fail if:

XSI [EILSEQ] — Input byte sequence does not form a valid character.

XSI [EINVAL] — There are insufficient arguments.

15852 **EXAMPLES**

15853 The call:

15854
```
int i, n; float x; char name[50];
```
15855
```
n = wscanf(L"%d%f%s", &i, &x, name);
```

15856 with the input line:

15857
```
25 54.32E-1 Hamster
```

15858 assigns to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* contains the string
15859 `"Hamster"`.

15860 The call:

15861
```
int i; float x; char name[50];
```
15862
```
(void) wscanf(L"%2d%f%*d %[0123456789]", &i, &x, name);
```

15863 with input:

15864
```
56789 0123 56a72
```

15865 assigns 56 to *i*, 789.0 to *x*, skip 0123, and place the string `"56\0"` in *name*. The next call to
15866 *getchar*( ) shall return the character `'a'`.

15867 **APPLICATION USAGE**

15868 In format strings containing the `'%'` form of conversion specifications, each argument in the
15869 argument list is used exactly once.

15870 **RATIONALE**

15871 None.

15872 **FUTURE DIRECTIONS**

15873 None.

15874 **SEE ALSO**

15875 *getwc*( ), *fwprintf*( ), *setlocale*( ), *wcstod*( ), *wcstol*( ), *wcstoul*( ), *wcrtomb*( ), the Base Definitions
15876 volume of IEEE Std 1003.1-200x, **<langinfo.h>**, **<stdio.h>**, **<wchar.h>**, the Base Definitions
15877 volume of IEEE Std 1003.1-200x, Chapter 7, Locale

15878 **CHANGE HISTORY**

15879 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
15880 (E).

15881 **Issue 6**

15882 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

15883 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

15884 - The prototypes for *fwscanf*( ) and *swscanf*( ) are updated.
15885 - The DESCRIPTION is updated. |
15886 - The `hh`, `ll`, `j`, `t`, and `z` length modifiers are added. |
15887 - The `a`, `A`, and `F` conversion characters are added. |

15888 The DESCRIPTION is updated to use the terms ‘‘conversion specifier’’ and ‘‘conversion
15889 specification’’ consistently.

15890 **NAME**

15891 gai_strerror — address and name information error description

15892 **SYNOPSIS**

15893 `#include <netdb.h>`

15894 `const char *gai_strerror(int ecode);`

15895 **DESCRIPTION**

15896 The *gai_strerror*( ) function shall return a text string describing an error value for the *getaddrinfo*( )
15897 and *getnameinfo*( ) functions listed in the **<netdb.h>** header.

15898 When the *ecode* argument is one of the following values listed in the **<netdb.h>** header:

15899 [EAI_AGAIN]
15900 [EAI_BADFLAGS]
15901 [EAI_FAIL]
15902 [EAI_FAMILY]
15903 [EAI_MEMORY]
15904 [EAI_NONAME]
15905 [EAI_SERVICE]
15906 [EAI_SOCKTYPE]
15907 [EAI_SYSTEM]

15908 the function return value shall point to a string describing the error. If the argument is not one
15909 of those values, the function shall return a pointer to a string whose contents indicate an
15910 unknown error.

15911 **RETURN VALUE**

15912 Upon successful completion, *gai_strerror*( ) shall return a pointer to an implementation-defined
15913 string.

15914 **ERRORS**

15915 No errors are defined.

15916 **EXAMPLES**

15917 None.

15918 **APPLICATION USAGE**

15919 None.

15920 **RATIONALE**

15921 None.

15922 **FUTURE DIRECTIONS**

15923 None.

15924 **SEE ALSO**

15925 *getaddrinfo*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

15926 **CHANGE HISTORY**

15927 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

15928 The Open Group Base Resolution bwg2001-009 is applied, which changes the return type from
15929 **char *** to **const char ***. This is for coordination with the IPnG Working Group.

15930 **NAME**

15931 gcvt — convert a floating-point number to a string (**LEGACY**)

15932 **SYNOPSIS**

15933 XSI
```
#include <stdlib.h>

char *gcvt(double value, int ndigit, char *buf);
```
15934

15935

15936 **DESCRIPTION**

15937 Refer to *ecvt*( ).

15938 **NAME**

15939 getaddrinfo — get address information

15940 **SYNOPSIS**

15941 ```
#include <sys/socket.h>
```

15942 ```
#include <netdb.h>
```

15943 ```
int getaddrinfo(const char *restrict nodename,
```

15944 ```
    const char *restrict servname,
```

15945 ```
    const struct addrinfo *restrict hints,
```

15946 ```
    struct addrinfo **restrict res);
```

15947 **DESCRIPTION**

15948 Refer to *freeaddrinfo*( ).

15949 **NAME**

15950 getc — get a byte from a stream

15951 **SYNOPSIS**

15952 `#include <stdio.h>`

15953 `int getc(FILE *stream);`

15954 **DESCRIPTION**

15955 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15956 conflict between the requirements described here and the ISO C standard is unintentional. This
15957 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

15958 The *getc*( ) function shall be equivalent to *fgetc*( ), except that if it is implemented as a macro it
15959 may evaluate *stream* more than once, so the argument should never be an expression with side
15960 effects.

15961 **RETURN VALUE**

15962 Refer to *fgetc*( ).

15963 **ERRORS**

15964 Refer to *fgetc*( ).

15965 **EXAMPLES**

15966 None.

15967 **APPLICATION USAGE**

15968 If the integer value returned by *getc*( ) is stored into a variable of type **char** and then compared
15969 against the integer constant EOF, the comparison may never succeed, because sign-extension of
15970 a variable of type **char** on widening to integer is implementation-defined.

15971 Since it may be implemented as a macro, *getc*( ) may treat incorrectly a *stream* argument with |
15972 side effects. In particular, *getc*(*f*++) does not necessarily work as expected. Therefore, use of this
15973 function should be preceded by `"#undef getc"` in such situations; *fgetc*( ) could also be used.

15974 **RATIONALE**

15975 None.

15976 **FUTURE DIRECTIONS**

15977 None.

15978 **SEE ALSO**

15979 *fgetc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

15980 **CHANGE HISTORY**

15981 First released in Issue 1. Derived from Issue 1 of the SVID.

15982 **NAME**

15983 getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked — stdio with explicit client
15984 locking

15985 **SYNOPSIS**

15986 TSF
```
#include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```
15991

15992 **DESCRIPTION**

15993 Versions of the functions *getc*(), *getchar*(), *putc*(), and *putchar*() respectively named
15994 *getc_unlocked*(), *getchar_unlocked*(), *putc_unlocked*(), and *putchar_unlocked*() shall be provided |
15995 which are functionally equivalent to the original versions, with the exception that they are not |
15996 required to be implemented in a thread-safe manner. They may only safely be used within a
15997 scope protected by *flockfile*() (or *ftrylockfile*()) and *funlockfile*(). These functions may safely be
15998 used in a multi-threaded program if and only if they are called while the invoking thread owns
15999 the (**FILE ***) object, as is the case after a successful call of the *flockfile*() or *ftrylockfile*() functions.

16000 **RETURN VALUE**

16001 See *getc*(), *getchar*(), *putc*(), and *putchar*().

16002 **ERRORS**

16003 See *getc*(), *getchar*(), *putc*(), and *putchar*(). |

16004 **EXAMPLES**

16005 None.

16006 **APPLICATION USAGE**

16007 Since they may be implemented as macros, *getc_unlocked*() and *putc_unlocked*() may treat |
16008 incorrectly a stream argument with side effects. In particular, *getc_unlocked*(*f++) and
16009 *putc_unlocked*(*f++) do not necessarily work as expected. Therefore, use of these functions in
16010 such situations should be preceded by the following statement as appropriate:

```
16011 #undef getc_unlocked
16012 #undef putc_unlocked
```

16013 **RATIONALE**

16014 Some I/O functions are typically implemented as macros for performance reasons (for example,
16015 *putc*() and *getc*()). For safety, they need to be synchronized, but it is often too expensive to
16016 synchronize on every character. Nevertheless, it was felt that the safety concerns were more
16017 important; consequently, the *getc*(), *getchar*(), *putc*(), and *putchar*() functions are required to be
16018 thread-safe. However, unlocked versions are also provided with names that clearly indicate the
16019 unsafe nature of their operation but can be used to exploit their higher performance. These
16020 unlocked versions can be safely used only within explicitly locked program regions, using
16021 exported locking primitives. In particular, a sequence such as:

```
16022 flockfile(fileptr);
16023 putc_unlocked('1', fileptr);
16024 putc_unlocked('\n', fileptr);
16025 fprintf(fileptr, "Line 2\n");
16026 funlockfile(fileptr);
```

16027 is permissible, and results in the text sequence:

```
1
Line 2
```

being printed without being interspersed with output from other threads.

It would be wrong to have the standard names such as *getc*(), *putc*(), and so on, map to the ''faster, but unsafe'' rather than the ''slower, but safe'' versions. In either case, you would still want to inspect all uses of *getc*(), *putc*(), and so on, by hand when converting existing code. Choosing the safe bindings as the default, at least, results in correct code and maintains the ''atomicity at the function'' invariant. To do otherwise would introduce gratuitous synchronization errors into converted code. Other routines that modify the *stdio* (**FILE ***) structures or buffers are also safely synchronized.

Note that there is no need for functions of the form *getc_locked*(), *putc_locked*(), and so on, since this is the functionality of *getc*(), *putc*(), *et al*. It would be inappropriate to use a feature test macro to switch a macro definition of *getc*() between *getc_locked*() and *getc_unlocked*(), since the ISO C standard requires an actual function to exist, a function whose behavior could not be changed by the feature test macro. Also, providing both the *xxx_locked*() and *xxx_unlocked*() forms leads to the confusion of whether the suffix describes the behavior of the function or the circumstances under which it should be used.

Three additional routines, *flockfile*(), *ftrylockfile*(), and *funlockfile*() (which may be macros), are provided to allow the user to delineate a sequence of I/O statements that are executed synchronously.

The *ungetc*() function is infrequently called relative to the other functions/macros so no unlocked variation is needed.

FUTURE DIRECTIONS

None.

SEE ALSO

getc(), *getchar*(), *putc*(), *putchar*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

These functions are marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U030/2 is applied, adding APPLICATION USAGE describing how applications should be written to avoid the case when the functions are implemented as macros.

16062 **NAME**

16063 getchar — get a byte from a stdin stream

16064 **SYNOPSIS**

16065
```
#include <stdio.h>
```

16066
```
int getchar(void);
```

16067 **DESCRIPTION**

16068 CX The functionality described on this reference page is aligned with the ISO C standard. Any
16069 conflict between the requirements described here and the ISO C standard is unintentional. This
16070 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

16071 The *getchar*( ) function shall be equivalent to *getc*(*stdin*).

16072 **RETURN VALUE**

16073 Refer to *fgetc*( ).

16074 **ERRORS**

16075 Refer to *fgetc*( ).

16076 **EXAMPLES**

16077 None.

16078 **APPLICATION USAGE**

16079 If the integer value returned by *getchar*( ) is stored into a variable of type **char** and then
16080 compared against the integer constant EOF, the comparison may never succeed, because sign-
16081 extension of a variable of type **char** on widening to integer is implementation-defined.

16082 **RATIONALE**

16083 None.

16084 **FUTURE DIRECTIONS**

16085 None.

16086 **SEE ALSO**

16087 *getc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

16088 **CHANGE HISTORY**

16089 First released in Issue 1. Derived from Issue 1 of the SVID.

16090 **NAME**

16091 getchar_unlocked — stdio with explicit client locking

16092 **SYNOPSIS**

16093 TSF
```
#include <stdio.h>
```

16094
```
int getchar_unlocked(void);
```

16095

16096 **DESCRIPTION**

16097 Refer to *getc_unlocked*( ).

16098 **NAME**

16099 getcontext, setcontext — get and set current user context

16100 **SYNOPSIS**

16101 XSI
```
#include <ucontext.h>

int getcontext(ucontext_t *ucp);
int setcontext(const ucontext_t *ucp);
```
16102
16103
16104

16105 **DESCRIPTION**

16106 The *getcontext*( ) function shall initialize the structure pointed to by *ucp* to the current user
16107 context of the calling thread. The **ucontext_t** type that *ucp* points to defines the user context and
16108 includes the contents of the calling thread's machine registers, the signal mask, and the current
16109 execution stack.

16110 The *setcontext*( ) function shall restore the user context pointed to by *ucp*. A successful call to
16111 *setcontext*( ) shall not return; program execution resumes at the point specified by the *ucp*
16112 argument passed to *setcontext*( ). The *ucp* argument should be created either by a prior call to
16113 *getcontext*( ) or *makecontext*( ), or by being passed as an argument to a signal handler. If the *ucp*
16114 argument was created with *getcontext*( ), program execution continues as if the corresponding
16115 call of *getcontext*( ) had just returned. If the *ucp* argument was created with *makecontext*( ),
16116 program execution continues with the function passed to *makecontext*( ). When that function
16117 returns, the thread shall continue as if after a call to *setcontext*( ) with the *ucp* argument that was
16118 input to *makecontext*( ). If the *uc_link* member of the **ucontext_t** structure pointed to by the *ucp*
16119 argument is equal to 0, then this context is the main context, and the thread shall exit when this
16120 context returns. The effects of passing a *ucp* argument obtained from any other source are
16121 unspecified.

16122 **RETURN VALUE**

16123 Upon successful completion, *setcontext*( ) shall not return and *getcontext*( ) shall return 0;
16124 otherwise, a value of –1 shall be returned.

16125 **ERRORS**

16126 No errors are defined.

16127 **EXAMPLES**

16128 Refer to *makecontext*( ). |

16129 **APPLICATION USAGE**

16130 When a signal handler is executed, the current user context is saved and a new context is
16131 created. If the thread leaves the signal handler via *longjmp*( ), then it is unspecified whether the
16132 context at the time of the corresponding *setjmp*( ) call is restored and thus whether future calls to
16133 *getcontext*( ) provide an accurate representation of the current context, since the context restored
16134 by *longjmp*( ) does not necessarily contain all the information that *setcontext*( ) requires. Signal
16135 handlers should use *siglongjmp*( ) or *setcontext*( ) instead.

16136 Conforming applications should not modify or access the *uc_mcontext* member of **ucontext_t**. A |
16137 conforming application cannot assume that context includes any process-wide static data, |
16138 possibly including *errno*. Users manipulating contexts should take care to handle these
16139 explicitly when required.

16140 Use of contexts to create alternate stacks is not defined by this volume of IEEE Std 1003.1-200x.

16141 **RATIONALE**

16142 None.

16143 **FUTURE DIRECTIONS**

16144 None.

16145 **SEE ALSO**

16146 *bsd_signal*( ), *makecontext*( ), *setjmp*( ), *sigaction*( ), *sigaltstack*( ), *sigprocmask*( ), *sigsetjmp*( ), the Base
16147 Definitions volume of IEEE Std 1003.1-200x, **<ucontext.h>**

16148 **CHANGE HISTORY**

16149 First released in Issue 4, Version 2.

16150 **Issue 5**

16151 Moved from X/OPEN UNIX extension to BASE.

16152 The following sentence was removed from the DESCRIPTION: ‘‘If the *ucp* argument was passed
16153 to a signal handler, program execution continues with the program instruction following the
16154 instruction interrupted by the signal.’’

16155 **NAME**

16156 getcwd — get the pathname of the current working directory |

16157 **SYNOPSIS**

16158
```
#include <unistd.h>
```

16159
```
char *getcwd(char *buf, size_t size);
```

16160 **DESCRIPTION**

16161 The *getcwd*( ) function shall place an absolute pathname of the current working directory in the |
16162 array pointed to by *buf*, and return *buf*. The pathname copied to the array shall contain no |
16163 components that are symbolic links. The *size* argument is the size in bytes of the character array |
16164 pointed to by the *buf* argument. If *buf* is a null pointer, the behavior of *getcwd*( ) is unspecified.

16165 **RETURN VALUE**

16166 Upon successful completion, *getcwd*( ) shall return the *buf* argument. Otherwise, *getcwd*( ) shall
16167 return a null pointer and set *errno* to indicate the error. The contents of the array pointed to by
16168 *buf* are then undefined.

16169 **ERRORS**

16170 The *getcwd*( ) function shall fail if:

16171 [EINVAL] The *size* argument is 0.

16172 [ERANGE] The size argument is greater than 0, but is smaller than the length of the |
16173 pathname +1. |

16174 The *getcwd*( ) function may fail if:

16175 [EACCES] Read or search permission was denied for a component of the pathname. |

16176 [ENOMEM] Insufficient storage space is available.

16177 **EXAMPLES**

16178 **Determining the Absolute Pathname of the Current Working Directory** |

16179 The following example returns a pointer to an array that holds the absolute pathname of the |
16180 current working directory. The pointer is returned in the *ptr* variable, which points to the *buf* |
16181 array where the pathname is stored. |

16182
```
#include <stdlib.h>
#include <unistd.h>
...
long size;
char *buf;
char *ptr;
```

16188
```
size = pathconf(".", _PC_PATH_MAX);
```

16189
```
if ((buf = (char *)malloc((size_t)size)) != NULL)
    ptr = getcwd(buf, (size_t)size);
...
```

16192 **APPLICATION USAGE**

16193 None.

16194 **RATIONALE**

16195 Since the maximum pathname length is arbitrary unless {PATH_MAX} is defined, an application |
16196 generally cannot supply a *buf* with *size* {{PATH_MAX}+1}.

16197 Having *getcwd*( ) take no arguments and instead use the *malloc*( ) function to produce space for
16198 the returned argument was considered. The advantage is that *getcwd*( ) knows how big the |
16199 working directory pathname is and can allocate an appropriate amount of space. But the |
16200 programmer would have to use the *free*( ) function to free the resulting object, or each use of
16201 *getcwd*( ) would further reduce the available memory. Also, *malloc*( ) and *free*( ) are used nowhere
16202 else in this volume of IEEE Std 1003.1-200x. Finally, *getcwd*( ) is taken from the SVID where it has
16203 the two arguments used in this volume of IEEE Std 1003.1-200x.

16204 The older function *getwd*( ) was rejected for use in this context because it had only a buffer
16205 argument and no *size* argument, and thus had no way to prevent overwriting the buffer, except
16206 to depend on the programmer to provide a large enough buffer.

16207 On some implementations, if *buf* is a null pointer, *getcwd*( ) may obtain *size* bytes of memory
16208 using *malloc*( ). In this case, the pointer returned by *getcwd*( ) may be used as the argument in a
16209 subsequent call to *free*( ). Invoking *getcwd*( ) with *buf* as a null pointer is not recommended in |
16210 conforming applications. |

16211 If a program is operating in a directory where some (grand)parent directory does not permit
16212 reading, *getcwd*( ) may fail, as in most implementations it must read the directory to determine
16213 the name of the file. This can occur if search, but not read, permission is granted in an
16214 intermediate directory, or if the program is placed in that directory by some more privileged
16215 process (for example, login). Including the [EACCES] error condition makes the reporting of the
16216 error consistent and warns the application writer that *getcwd*( ) can fail for reasons beyond the
16217 control of the application writer or user. Some implementations can avoid this occurrence (for
16218 example, by implementing *getcwd*( ) using *pwd*, where *pwd* is a set-user-root process), thus the |
16219 error was made optional. Since this volume of IEEE Std 1003.1-200x permits the addition of other |
16220 errors, this would be a common addition and yet one that applications could not be expected to |
16221 deal with without this addition. |

16222 **FUTURE DIRECTIONS**

16223 None.

16224 **SEE ALSO**

16225 *malloc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

16226 **CHANGE HISTORY**

16227 First released in Issue 1. Derived from Issue 1 of the SVID.

16228 **Issue 6**

16229 The following new requirements on POSIX implementations derive from alignment with the
16230 Single UNIX Specification:

16231 - The [ENOMEM] optional error condition is added.

16232 **NAME**

16233 getdate — convert user format date and time

16234 **SYNOPSIS**

16235 XSI
```
#include <time.h>

struct tm *getdate(const char *string);
```
16236

16237

16238 **DESCRIPTION**

16239 The *getdate*( ) function shall convert a string representation of a date or time into a broken-down
16240 time.

16241 The external variable or macro *getdate_err* is used by *getdate*( ) to return error values.

16242 Templates are used to parse and interpret the input string. The templates are contained in a text
16243 file identified by the environment variable *DATEMSK*. The *DATEMSK* variable should be set to
16244 indicate the full pathname of the file that contains the templates. The first line in the template
16245 that matches the input specification is used for interpretation and conversion into the internal
16246 time format.

16247 The following conversion specifications shall be supported:

16248 `%%` Equivalent to %.

16249 `%a` Abbreviated weekday name.

16250 `%A` Full weekday name.

16251 `%b` Abbreviated month name.

16252 `%B` Full month name.

16253 `%c` Locale's appropriate date and time representation.

16254 `%C` Century number [00,99]; leading zeros are permitted but not required.

16255 `%d` Day of month [01,31]; the leading 0 is optional.

16256 `%D` Date as `%m/%d/%y`.

16257 `%e` Equivalent to `%d`.

16258 `%h` Abbreviated month name.

16259 `%H` Hour [00,23].

16260 `%I` Hour [01,12].

16261 `%m` Month number [01,12].

16262 `%M` Minute [00,59].

16263 `%n` Equivalent to <newline>.

16264 `%p` Locale's equivalent of either AM or PM.

16265 `%r` The locale's appropriate representation of time in AM and PM notation. In the POSIX
16266 locale, this shall be equivalent to `%I:%M:%S %p`.

16267 `%R` Time as `%H:%M`.

16268 `%S` Seconds [00,60]. The range goes to 60 (rather than stopping at 59) to allow positive leap
16269 seconds to be expressed. Since leap seconds cannot be predicted by any algorithm, leap
16270 second data must come from some external source.

16271 %t Equivalent to <tab>.

16272 %T Time as %H:%M:%S.

16273 %w Weekday number (Sunday = [0,6]).

16274 %x Locale's appropriate date representation.

16275 %X Locale's appropriate time representation.

16276 %y Year within century. When a century is not otherwise specified, values in the range
16277 [69,99] shall refer to years 1969 to 1999 inclusive, and values in the range [00,68] shall
16278 refer to years 2000 to 2068 inclusive.

16279 **Note:** It is expected that in a future version of IEEE Std 1003.1-200x the default century
16280 inferred from a 2-digit year will change. (This would apply to all commands
16281 accepting a 2-digit year as input.)

16282 %Y Year as "ccyy" (for example, 2001).

16283 %Z Timezone name or no characters if no timezone exists. If the timezone supplied by %Z is
16284 not the timezone that *getdate*( ) expects, an invalid input specification error shall result.
16285 The *getdate*( ) function calculates an expected timezone based on information supplied
16286 to the function (such as the hour, day, and month).

16287 The match between the template and input specification performed by *getdate*( ) shall be case-
16288 insensitive.

16289 The month and weekday names can consist of any combination of upper and lowercase letters.
16290 The process can request that the input date or time specification be in a specific language by
16291 setting the *LC_TIME* category (see *setlocale*( )).

16292 Leading 0s are not necessary for the descriptors that allow leading 0s. However, at most two
16293 digits are allowed for those descriptors, including leading 0s. Extra whitespace in either the
16294 template file or in *string* shall be ignored.

16295 The results are undefined if the conversion specifications %c, %x, and %X include unsupported
16296 conversion specifications.

16297 The following rules apply for converting the input specification into the internal format:

16298 • If %Z is being scanned, then *getdate*( ) shall initialize the broken-down time to be the current
16299 time in the scanned timezone. Otherwise, it shall initialize the broken-down time based on
16300 the current local time as if *localtime*( ) had been called.

16301 • If only the weekday is given, the day chosen shall be the day, starting with today and moving
16302 into the future, which first matches the named day.

16303 • If only the month (and no year) is given, the month chosen shall be the month, starting with
16304 the current month and moving into the future, which first matches the named month. The
16305 first day of the month shall be assumed if no day is given.

16306 • If no hour, minute, and second are given the current hour, minute, and second shall be
16307 assumed.

16308 • If no date is given, the hour chosen shall be the hour, starting with the current hour and
16309 moving into the future, which first matches the named hour.

16310 If a conversion specification in the DATEMSK file does not correspond to one of the conversion
16311 specifications above, the behavior is unspecified.

16312 The *getdate*( ) function need not be reentrant. A function that is not required to be reentrant is not
16313 required to be thread-safe.

16314 **RETURN VALUE**

16315 Upon successful completion, *getdate*() shall return a pointer to a **struct tm**. Otherwise, it shall
16316 return a null pointer and set *getdate_err* to indicate the error.

16317 **ERRORS**

16318 The *getdate*() function shall fail in the following cases, setting *getdate_err* to the value shown in
16319 the list below. Any changes to *errno* are unspecified.

16320 1. The *DATEMSK* environment variable is null or undefined.

16321 2. The template file cannot be opened for reading.

16322 3. Failed to get file status information.

16323 4. The template file is not a regular file.

16324 5. An I/O error is encountered while reading the template file.

16325 6. Memory allocation failed (not enough memory available).

16326 7. There is no line in the template that matches the input.

16327 8. Invalid input specification. For example, February 31; or a time is specified that cannot be
16328 represented in a **time_t** (representing the time in seconds since the Epoch).

16329 **EXAMPLES**

16330 1. The following example shows the possible contents of a template:

```
16331 %m
16332 %A %B %d, %Y, %H:%M:%S
16333 %A
16334 %B
16335 %m/%d/%y %I %p
16336 %d,%m,%Y %H:%M
16337 at %A the %dst of %B in %Y
16338 run job at %I %p,%B %dnd
16339 %A den %d. %B %Y %H.%M Uhr
```

16340 2. The following are examples of valid input specifications for the template in Example 1:

```
16341 getdate("10/1/87 4 PM");
16342 getdate("Friday");
16343 getdate("Friday September 18, 1987, 10:30:30");
16344 getdate("24,9,1986 10:30");
16345 getdate("at monday the 1st of december in 1986");
16346 getdate("run job at 3 PM, december 2nd");
```

16347 If the *LC_TIME* category is set to a German locale that includes *freitag* as a weekday name
16348 and *oktober* as a month name, the following would be valid:

```
16349 getdate("freitag den 10. oktober 1986 10.30 Uhr");
```

16350 3. The following example shows how local date and time specification can be defined in the
16351 template:

Invocation	Line in Template
`getdate("11/27/86")`	`%m/%d/%y`
`getdate("27.11.86")`	`%d.%m.%y`
`getdate("86-11-27")`	`%y-%m-%d`
`getdate("Friday 12:00:00")`	`%A %H:%M:%S`

4. The following examples help to illustrate the above rules assuming that the current date is Mon Sep 22 12:19:47 EDT 1986 and the *LC_TIME* category is set to the default C locale:

Input	Line in Template	Date
`Mon`	`%a`	Mon Sep 22 12:19:47 EDT 1986
`Sun`	`%a`	Sun Sep 28 12:19:47 EDT 1986
`Fri`	`%a`	Fri Sep 26 12:19:47 EDT 1986
`September`	`%B`	Mon Sep 1 12:19:47 EDT 1986
`January`	`%B`	Thu Jan 1 12:19:47 EST 1987
`December`	`%B`	Mon Dec 1 12:19:47 EST 1986
`Sep Mon`	`%b %a`	Mon Sep 1 12:19:47 EDT 1986
`Jan Fri`	`%b %a`	Fri Jan 2 12:19:47 EST 1987
`Dec Mon`	`%b %a`	Mon Dec 1 12:19:47 EST 1986
`Jan Wed 1989`	`%b %a %Y`	Wed Jan 4 12:19:47 EST 1989
`Fri 9`	`%a %H`	Fri Sep 26 09:00:00 EDT 1986
`Feb 10:30`	`%b %H:%S`	Sun Feb 1 10:00:30 EST 1987
`10:30`	`%H:%M`	Tue Sep 23 10:30:00 EDT 1986
`13:30`	`%H:%M`	Mon Sep 22 13:30:00 EDT 1986

APPLICATION USAGE

Although historical versions of *getdate*() did not require that **<time.h>** declare the external variable *getdate_err*, this volume of IEEE Std 1003.1-200x does require it. The standard developers encourage applications to remove declarations of *getdate_err* and instead incorporate the declaration by including **<time.h>**.

Applications should use `%Y` (4-digit years) in preference to `%y` (2-digit years).

RATIONALE

In standard locales, the conversion specifications `%c`, `%x`, and `%X` do not include unsupported conversion specifiers and so the text regarding results being undefined is not a problem in that case.

FUTURE DIRECTIONS

None.

SEE ALSO

ctime(), *localtime*(), *setlocale*(), *strftime*(), *times*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The last paragraph of the DESCRIPTION is added.

The `%C` conversion specification is added, and the exact meaning of the `%y` conversion specification is clarified in the DESCRIPTION.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

The %R conversion specifications is changed to follow historical practice.

Issue 6

The DESCRIPTION is updated to refer to ‘‘seconds since the Epoch’’ rather than ‘‘seconds since 00:00:00 UTC (Coordinated Universal Time), January 1 1970’’ for consistency with other *time* functions.

The description of %S is updated so that the valid range [00,60] rather than [00,61]. |

The DESCRIPTION is updated to refer to conversion specifications instead of field descriptors for consistency with other functions. |

16407 **NAME**

16408 getegid — get the effective group ID

16409 **SYNOPSIS**

16410
```
#include <unistd.h>
```

16411
```
gid_t getegid(void);
```

16412 **DESCRIPTION**

16413 The *getegid*( ) function shall return the effective group ID of the calling process.

16414 **RETURN VALUE**

16415 The *getegid*( ) function shall always be successful and no return value is reserved to indicate an |
16416 error.

16417 **ERRORS**

16418 No errors are defined.

16419 **EXAMPLES**

16420 None.

16421 **APPLICATION USAGE**

16422 None.

16423 **RATIONALE**

16424 None.

16425 **FUTURE DIRECTIONS**

16426 None.

16427 **SEE ALSO**

16428 *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base
16429 Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

16430 **CHANGE HISTORY**

16431 First released in Issue 1. Derived from Issue 1 of the SVID.

16432 **Issue 6**

16433 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

16434 The following new requirements on POSIX implementations derive from alignment with the
16435 Single UNIX Specification:

16436 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
16437 required for conforming implementations of previous POSIX specifications, it was not
16438 required for UNIX applications.

16439 **NAME**

16440 getenv — get value of an environment variable

16441 **SYNOPSIS**

16442
```
#include <stdlib.h>
```

16443
```
char *getenv(const char *name);
```

16444 **DESCRIPTION**

16445 CX The functionality described on this reference page is aligned with the ISO C standard. Any
16446 conflict between the requirements described here and the ISO C standard is unintentional. This
16447 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

16448 The *getenv*() function shall search the environment of the calling process (see the Base
16449 Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables) for the
16450 environment variable *name* if it exists and return a pointer to the value of the environment
16451 variable. If the specified environment variable cannot be found, a null pointer shall be returned.
16452 The application shall ensure that it does not modify the string pointed to by the *getenv*()
16453 function.

16454 CX The string pointed to may be overwritten by a subsequent call to *getenv*(), *setenv*(), or *unsetenv*(),
16455 but shall not be overwritten by a call to any other function in this volume of
16456 IEEE Std 1003.1-200x.

16457 CX If the application modifies *environ* or the pointers to which it points, the behavior of *getenv*() is |
16458 undefined.

16459 The *getenv*() function need not be reentrant. A function that is not required to be reentrant is not
16460 required to be thread-safe.

16461 **RETURN VALUE**

16462 Upon successful completion, *getenv*() shall return a pointer to a string containing the *value* for
16463 the specified *name*. If the specified name cannot be found in the environment of the calling
16464 process, a null pointer shall be returned.

16465 The return value from *getenv*() may point to static data which may be overwritten by
16466 CX subsequent calls to *getenv*(), *setenv*(), or *unsetenv*().

16467 XSI On XSI-conformant systems, the return value from *getenv*() may point to static data which may
16468 also be overwritten by subsequent calls to *putenv*().

16469 **ERRORS**

16470 No errors are defined.

16471 **EXAMPLES**

16472 **Getting the Value of an Environment Variable**

16473 The following example gets the value of the *HOME* environment variable.

16474
```
#include <stdlib.h>
```
16475
```
...
```
16476
```
const char *name = "HOME";
char *value;
```
16477

16478
```
value = getenv(name);
```

16479 **APPLICATION USAGE**

16480 None.

16481 **RATIONALE**

16482 The *clearenv*( ) function was considered but rejected. The *putenv*( ) function has now been
16483 included for alignment with the Single UNIX Specification.

16484 The *getenv*( ) function is inherently not reentrant because it returns a value pointing to static
16485 data.

16486 Conforming applications are required not to modify *environ* directly, but to use only the
16487 functions described here to manipulate the process environment as an abstract object. Thus, the
16488 implementation of the environment access functions has complete control over the data
16489 structure used to represent the environment (subject to the requirement that *environ* be
16490 maintained as a list of strings with embedded equal signs for applications that wish to scan the
16491 environment). This constraint allows the implementation to properly manage the memory it
16492 allocates, either by using allocated storage for all variables (copying them on the first invocation
16493 of *setenv*( ) or *unsetenv*( )), or keeping track of which strings are currently in allocated space and
16494 which are not, via a separate table or some other means. This enables the implementation to free
16495 any allocated space used by strings (and perhaps the pointers to them) stored in *environ* when
16496 *unsetenv*( ) is called. A C runtime start-up procedure (that which invokes *main*( ) and perhaps
16497 initializes *environ*) can also initialize a flag indicating that none of the environment has yet been
16498 copied to allocated storage, or that the separate table has not yet been initialized.

16499 In fact, for higher performance of *getenv*( ), the implementation could also maintain a separate
16500 copy of the environment in a data structure that could be searched much more quickly (such as
16501 an indexed hash table, or a binary tree), and update both it and the linear list at *environ* when
16502 *setenv*( ) or *unsetenv*( ) is invoked.

16503 Performance of *getenv*( ) can be important for applications which have large numbers of
16504 environment variables. Typically, applications like this use the environment as a resource |
16505 database of user-configurable parameters. The fact that these variables are in the user's shell |
16506 environmentt usually means that any other program that uses environment variables (such as *ls*,
16507 which attempts to use *COLUMNS*, or really almost any utility (*LANG*, *LC_ALL*, and so on) is
16508 similarly slowed down by the linear search through the variables.

16509 An implementation that maintains separate data structures, or even one that manages the
16510 memory it consumes, is not currently required as it was thought it would reduce consensus
16511 among implementors who do not want to change their historical implementations.

16512 The POSIX Threads Extension states that multi-threaded applications must not modify *environ*
16513 directly, and that IEEE Std 1003.1-200x is providing functions which such applications can use in
16514 the future to manipulate the environment in a thread-safe manner. Thus, moving away from
16515 application use of *environ* is desirable from that standpoint as well.

16516 **FUTURE DIRECTIONS**

16517 None.

16518 **SEE ALSO**

16519 *exec*, *putenv*( ), *setenv*( ), *unsetenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, |
16520 **<stdlib.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment
16521 Variables

16522 **CHANGE HISTORY**

16523 First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

The following changes were made to align with the IEEE P1003.1a draft standard:

- References added to the new *setenv*( ) and *unsetenv*( ) functions.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

16532 **NAME**

16533 geteuid — get the effective user ID

16534 **SYNOPSIS**

16535
```
#include <unistd.h>
```

16536
```
uid_t geteuid(void);
```

16537 **DESCRIPTION**

16538 The *geteuid*( ) function shall return the effective user ID of the calling process.

16539 **RETURN VALUE**

16540 The *geteuid*( ) function shall always be successful and no return value is reserved to indicate an |
16541 error.

16542 **ERRORS**

16543 No errors are defined.

16544 **EXAMPLES**

16545 None.

16546 **APPLICATION USAGE**

16547 None.

16548 **RATIONALE**

16549 None.

16550 **FUTURE DIRECTIONS**

16551 None.

16552 **SEE ALSO**

16553 *getegid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base
16554 Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

16555 **CHANGE HISTORY**

16556 First released in Issue 1. Derived from Issue 1 of the SVID.

16557 **Issue 6**

16558 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

16559 The following new requirements on POSIX implementations derive from alignment with the
16560 Single UNIX Specification:

16561 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
16562 required for conforming implementations of previous POSIX specifications, it was not
16563 required for UNIX applications.

16564 **NAME**

16565 getgid — get the real group ID

16566 **SYNOPSIS**

16567
```
#include <unistd.h>
```

16568
```
gid_t getgid(void);
```

16569 **DESCRIPTION**

16570 The *getgid*( ) function shall return the real group ID of the calling process.

16571 **RETURN VALUE**

16572 The *getgid*( ) function shall always be successful and no return value is reserved to indicate an |
16573 error.

16574 **ERRORS**

16575 No errors are defined.

16576 **EXAMPLES**

16577 None.

16578 **APPLICATION USAGE**

16579 None.

16580 **RATIONALE**

16581 None.

16582 **FUTURE DIRECTIONS**

16583 None.

16584 **SEE ALSO**

16585 *getegid*( ), *geteuid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base
16586 Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

16587 **CHANGE HISTORY**

16588 First released in Issue 1. Derived from Issue 1 of the SVID.

16589 **Issue 6**

16590 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

16591 The following new requirements on POSIX implementations derive from alignment with the
16592 Single UNIX Specification:

16593 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
16594 required for conforming implementations of previous POSIX specifications, it was not
16595 required for UNIX applications.

16596 **NAME**

16597 getgrent — get the group database entry

16598 **SYNOPSIS**

16599 XSI
```
#include <grp.h>
```

16600
```
struct group *getgrent(void);
```

16601

16602 **DESCRIPTION**

16603 Refer to *endgrent*( ).

16604 **NAME**

16605 getgrgid, getgrgid_r — get group database entry for a group ID

16606 **SYNOPSIS**

16607 ```
#include <grp.h>
```

16608 ```
struct group *getgrgid(gid_t gid);
```

16609 TSF ```
int getgrgid_r(gid_t gid, struct group *grp, char *buffer,
```

16610 ```
    size_t bufsize, struct group **result);
```

16611

16612 **DESCRIPTION**

16613 The *getgrgid*( ) function shall search the group database for an entry with a matching *gid*.

16614 The *getgrgid*( ) function need not be reentrant. A function that is not required to be reentrant is
16615 not required to be thread-safe.

16616 TSF The *getgrgid_r*( ) function shall update the **group** structure pointed to by *grp* and store a pointer |
16617 to that structure at the location pointed to by *result*. The structure shall contain an entry from |
16618 the group database with a matching *gid*. Storage referenced by the group structure is allocated |
16619 from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
16620 maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX}
16621 *sysconf*( ) parameter. A NULL pointer shall be returned at the location pointed to by *result* on |
16622 error or if the requested entry is not found.

16623 **RETURN VALUE**

16624 Upon successful completion, *getgrgid*( ) shall return a pointer to a **struct group** with the structure
16625 defined in **<grp.h>** with a matching entry if one is found. The *getgrgid*( ) function shall return a
16626 null pointer if either the requested entry was not found, or an error occurred. On error, *errno*
16627 shall be set to indicate the error.

16628 The return value may point to a static area which is overwritten by a subsequent call to
16629 *getgrent*( ), *getgrgid*( ), or *getgrnam*( ).

16630 TSF If successful, the *getgrgid_r*( ) function shall return zero; otherwise, an error number shall be
16631 returned to indicate the error.

16632 **ERRORS**

16633 The *getgrgid*( ) and *getgrgid_r*( ) functions may fail if:

16634 [EIO] An I/O error has occurred.

16635 [EINTR] A signal was caught during *getgrgid*( ).

16636 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

16637 [ENFILE] The maximum allowable number of files is currently open in the system.

16638 TSF The *getgrgid_r*( ) function may fail if:

16639 TSF [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to
16640 be referenced by the resulting **group** structure.

EXAMPLES

Finding an Entry in the Group Database

The following example uses *getgrgid*() to search the group database for a group ID that was previously stored in a **stat** structure, then prints out the group name if it is found. If the group is not found, the program prints the numeric value of the group for the entry.

```
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>
...
struct stat statbuf;
struct group *grp;
...
if ((grp = getgrgid(statbuf.st_gid)) != NULL)
    printf(" %-8.8s", grp->gr_name);
else
    printf(" %-8d", statbuf.st_gid);
...
```

APPLICATION USAGE

Applications wishing to check for error situations should set *errno* to 0 before calling *getgrgid*(). If *errno* is set on return, an error occurred.

The *getgrgid_r*() function is thread-safe and shall return values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endgrent(), *getgrnam*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<grp.h>**, **<limits.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The *getgrgid_r*() function is included for alignment with the POSIX Threads Extension.

A note indicating that the *getgrgid*() function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *getgrgid_r*() function is marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION describing matching the *gid*.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from *bufsize* characters to bytes.

16695 **NAME**

16696 getgrnam, getgrnam_r — search group database for a name

16697 **SYNOPSIS**

16698 ```
#include <grp.h>
```

16699 ```
struct group *getgrnam(const char *name);
```

16700 TSF ```
int getgrnam_r(const char *name, struct group *grp, char *buffer,
```

16701 ```
    size_t bufsize, struct group **result);
```

16702

16703 **DESCRIPTION**

16704 The *getgrnam*( ) function shall search the group database for an entry with a matching *name*.

16705 The *getgrnam*( ) function need not be reentrant. A function that is not required to be reentrant is
16706 not required to be thread-safe.

16707 TSF The *getgrnam_r*( ) function shall update the **group** structure pointed to by *grp* and store a pointer
16708 to that structure at the location pointed to by *result*. The structure shall contain an entry from
16709 the group database with a matching *gid* or *name*. Storage referenced by the group structure is
16710 allocated from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
16711 maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX}
16712 *sysconf*( ) parameter. A NULL pointer is returned at the location pointed to by *result* on error or if
16713 the requested entry is not found.

16714 **RETURN VALUE**

16715 The *getgrnam*( ) function shall return a pointer to a **struct group** with the structure defined in
16716 **<grp.h>** with a matching entry if one is found. The *getgrnam*( ) function shall return a null
16717 pointer if either the requested entry was not found, or an error occurred. On error, *errno* shall be
16718 set to indicate the error.

16719 The return value may point to a static area which is overwritten by a subsequent call to
16720 *getgrent*( ), *getgrgid*( ), or *getgrnam*( ).

16721 TSF If successful, the *getgrnam_r*( ) function shall return zero; otherwise, an error number shall be
16722 returned to indicate the error.

16723 **ERRORS**

16724 The *getgrnam*( ) and *getgrnam_r*( ) functions may fail if:

16725 [EIO] An I/O error has occurred.

16726 [EINTR] A signal was caught during *getgrnam*( ).

16727 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

16728 [ENFILE] The maximum allowable number of files is currently open in the system.

16729 The *getgrnam_r*( ) function may fail if:

16730 TSF [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to
16731 be referenced by the resulting **group** structure.

EXAMPLES

None.

APPLICATION USAGE

Applications wishing to check for error situations should set *errno* to 0 before calling *getgrnam*(). If *errno* is set on return, an error occurred.

The *getgrnam_r*() function is thread-safe and shall return values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endgrent(), *getgrgid*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<grp.h>**, **<limits.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The *getgrnam_r*() function is included for alignment with the POSIX Threads Extension.

A note indicating that the *getgrnam*() function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *getgrnam_r*() function is marked as part of the Thread-Safe Functions option.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from *bufsize* characters to bytes.

16769 **NAME**

16770 getgroups — get supplementary group IDs

16771 **SYNOPSIS**

16772 `#include <unistd.h>`

16773 `int getgroups(int gidsetsize, gid_t grouplist[]);`

16774 **DESCRIPTION**

16775 The *getgroups*( ) function shall fill in the array *grouplist* with the current supplementary group |
16776 IDs of the calling process. It is implementation-defined whether *getgroups*( ) also returns the |
16777 effective group ID in the *grouplist* array.

16778 The *gidsetsize* argument specifies the number of elements in the array *grouplist*. The actual |
16779 number of group IDs stored in the array shall be returned. The values of array entries with |
16780 indices greater than or equal to the value returned are undefined. |

16781 If *gidsetsize* is 0, *getgroups*( ) shall return the number of group IDs that it would otherwise return
16782 without modifying the array pointed to by *grouplist*.

16783 If the effective group ID of the process is returned with the supplementary group IDs, the value
16784 returned shall always be greater than or equal to one and less than or equal to the value of
16785 {NGROUPS_MAX}+1.

16786 **RETURN VALUE**

16787 Upon successful completion, the number of supplementary group IDs shall be returned. A
16788 return value of –1 indicates failure and *errno* shall be set to indicate the error.

16789 **ERRORS**

16790 The *getgroups*( ) function shall fail if:

16791 [EINVAL] The *gidsetsize* argument is non-zero and less than the number of group IDs
16792 that would have been returned.

16793 **EXAMPLES**

16794 **Getting the Supplementary Group IDs of the Calling Process**

16795 The following example places the current supplementary group IDs of the calling process into
16796 the *group* array.

```
16797 #include <sys/types.h>
16798 #include <unistd.h>
16799 ...
16800 gid_t *group;
16801 int nogroups;
16802 long ngroups_max;
16803 ngroups_max = sysconf(_SC_NGROUPS_MAX) + 1;
16804 group = (gid_t *)malloc(ngroups_max *sizeof(gid_t));
16805 ngroups = getgroups(ngroups_max, group);
```

16806 **APPLICATION USAGE**

16807 None.

16808 **RATIONALE**

16809 The related function *setgroups*( ) is a privileged operation and therefore is not covered by this
16810 volume of IEEE Std 1003.1-200x.

As implied by the definition of supplementary groups, the effective group ID may appear in the array returned by *getgroups*( ) or it may be returned only by *getegid*( ). Duplication may exist, but the application needs to call *getegid*( ) to be sure of getting all of the information. Various implementation variations and administrative sequences cause the set of groups appearing in the result of *getgroups*( ) to vary in order and as to whether the effective group ID is included, even when the set of groups is the same (in the mathematical sense of ''set''). (The history of a process and its parents could affect the details of result.)

Applications writers should note that {NGROUPS_MAX} is not necessarily a constant on all implementations.

FUTURE DIRECTIONS

None.

SEE ALSO

getegid( ), *setgid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- A return value of 0 is not permitted, because {NGROUPS_MAX} cannot be 0. This is a FIPS requirement.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Explanation added that the effective group ID may be included in the supplementary group list.

16843 **NAME**

16844 gethostbyaddr, gethostbyname — network host database functions

16845 **SYNOPSIS**

16846 `#include <netdb.h>`

16847 OB
```
struct hostent *gethostbyaddr(const void *addr, socklen_t len,
    int type);
struct hostent *gethostbyname(const char *name);
```
16850

16851 **DESCRIPTION**

16852 These functions shall retrieve information about hosts. This information is considered to be
16853 stored in a database that can be accessed sequentially or randomly. Implementation of this
16854 database is unspecified.

16855 **Note:** In many cases it is implemented by the Domain Name System, as documented in RFC 1034,
16856 RFC 1035, and RFC 1886.

16857 Entries shall be returned in **hostent** structures.

16858 The *gethostbyaddr*( ) function shall return an entry containing addresses of address family *type* for
16859 the host with address *addr*. The *len* argument contains the length of the address pointed to by
16860 *addr*. The *gethostbyaddr*( ) function need not be reentrant. A function that is not required to be
16861 reentrant is not required to be thread-safe.

16862 The *gethostbyname*( ) function shall return an entry containing addresses of address family
16863 AF_INET for the host with name *name*. The *gethostbyname*( ) function need not be reentrant. A
16864 function that is not required to be reentrant is not required to be thread-safe.

16865 The *addr* argument of *gethostbyaddr*( ) shall be an **in_addr** structure when *type* is AF_INET. It
16866 contains a binary format (that is, not null-terminated) address in network byte order. The
16867 *gethostbyaddr*( ) function is not guaranteed to return addresses of address families other than
16868 AF_INET, even when such addresses exist in the database.

16869 If *gethostbyaddr*( ) returns successfully, then the *h_addrtype* field in the result shall be the same as
16870 the *type* argument that was passed to the function, and the *h_addr_list* field shall list a single
16871 address that is a copy of the *addr* argument that was passed to the function.

16872 The *name* argument of *gethostbyname*( ) shall be a node name; the behavior of *gethostbyname*( )
16873 when passed a numeric address string is unspecified. For IPv4, a numeric address string shall be
16874 in the dotted-decimal notation described in *inet_addr*( ).

16875 If *name* is not a numeric address string and is an alias for a valid host name, then *gethostbyname*( )
16876 shall return information about the host name to which the alias refers, and *name* shall be
16877 included in the list of aliases returned.

16878 **RETURN VALUE**

16879 Upon successful completion, these functions shall return a pointer to a **hostent** structure if the
16880 requested entry was found, and a null pointer if the end of the database was reached or the
16881 requested entry was not found.

16882 Upon unsuccessful completion, *gethostbyaddr*( ) and *gethostbyname*( ) shall set *h_errno* to indicate
16883 the error.

16884 **ERRORS**

16885 These functions shall fail in the following cases. The *gethostbyaddr*( ) and *gethostbyname*( )
16886 functions shall set *h_errno* to the value shown in the list below. Any changes to *errno* are
16887 unspecified.

16888 [HOST_NOT_FOUND]
16889 No such host is known.

16890 [NO_DATA] The server recognized the request and the name, but no address is available.
16891 Another type of request to the name server for the domain might return an
16892 answer.

16893 [NO_RECOVERY]
16894 An unexpected server failure occurred which cannot be recovered.

16895 [TRY_AGAIN] A temporary and possibly transient error occurred, such as a failure of a
16896 server to respond.

16897 **EXAMPLES**

16898 None.

16899 **APPLICATION USAGE**

16900 The *gethostbyaddr*( ) and *gethostbyname*( ) functions may return pointers to static data, which may
16901 be overwritten by subsequent calls to any of these functions.

16902 The *getaddrinfo*( ) and *getnameinfo*( ) functions are preferred over the *gethostbyaddr*( ) and
16903 *gethostbyname*( ) functions.

16904 **RATIONALE**

16905 None.

16906 **FUTURE DIRECTIONS**

16907 The *gethostbyaddr*( ) and *gethostbyname*( ) functions may be withdrawn in a future version.

16908 **SEE ALSO**

16909 *endhostent*( ), *endservent*( ), *gai_strerror*( ), *getaddrinfo*( ), *h_errno*, *inet_addr*( ), the Base Definitions
16910 volume of IEEE Std 1003.1-200x, **<netdb.h>**

16911 **CHANGE HISTORY**

16912 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

16913 **NAME**

16914 gethostbyname — network host database functions

16915 **SYNOPSIS**

16916
```
#include <netdb.h>
```

16917 OB
```
struct hostent *gethostbyname(const char *name);
```

16918

16919 **DESCRIPTION**

16920 Refer to *gethostbyaddr*( ).

16921 **NAME**

16922 gethostent — network host database functions

16923 **SYNOPSIS**

16924 `#include <netdb.h>`

16925 `struct hostent *gethostent(void);`

16926 **DESCRIPTION**

16927 Refer to *endhostent*( ).

16928 **NAME**

16929 gethostid — get an identifier for the current host

16930 **SYNOPSIS**

16931 XSI
```
#include <unistd.h>
```
16932
```
long gethostid(void);
```
16933

16934 **DESCRIPTION**

16935 The *gethostid*( ) function shall retrieve a 32-bit identifier for the current host.

16936 **RETURN VALUE**

16937 Upon successful completion, *gethostid*( ) shall return an identifier for the current host.

16938 **ERRORS**

16939 No errors are defined.

16940 **EXAMPLES**

16941 None.

16942 **APPLICATION USAGE**

16943 This volume of IEEE Std 1003.1-200x does not define the domain in which the return value is
16944 unique.

16945 **RATIONALE**

16946 None.

16947 **FUTURE DIRECTIONS**

16948 None.

16949 **SEE ALSO**

16950 *random*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

16951 **CHANGE HISTORY**

16952 First released in Issue 4, Version 2.

16953 **Issue 5**

16954 Moved from X/OPEN UNIX extension to BASE.

16955 **NAME**

16956 gethostname — get name of current host

16957 **SYNOPSIS**

16958 `#include <unistd.h>`

16959 `int gethostname(char *name, size_t namelen);`

16960 **DESCRIPTION**

16961 The *gethostname*( ) function shall return the standard host name for the current machine. The
16962 *namelen* argument shall specify the size of the array pointed to by the *name* argument. The
16963 returned name shall be null-terminated, except that if *namelen* is an insufficient length to hold
16964 the host name, then the returned name shall be truncated and it is unspecified whether the
16965 returned name is null-terminated.

16966 Host names are limited to {HOST_NAME_MAX} bytes.

16967 **RETURN VALUE**

16968 Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned.

16969 **ERRORS**

16970 No errors are defined.

16971 **EXAMPLES**

16972 None.

16973 **APPLICATION USAGE**

16974 None.

16975 **RATIONALE**

16976 None.

16977 **FUTURE DIRECTIONS**

16978 None.

16979 **SEE ALSO**

16980 *gethostid*( ), *uname*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

16981 **CHANGE HISTORY**

16982 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

16983 The Open Group Base Resolution bwg2001-008 is applied, changing the *namelen* parameter from
16984 **socklen_t** to **size_t**.

16985 **NAME**

16986 getitimer, setitimer — get and set value of interval timer

16987 **SYNOPSIS**

16988 XSI
```
#include <sys/time.h>

int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *restrict value,
    struct itimerval *restrict ovalue);
```
16992

16993 **DESCRIPTION**

16994 The *getitimer*( ) function shall store the current value of the timer specified by *which* into the
16995 structure pointed to by *value*. The *setitimer*( ) function shall set the timer specified by *which* to
16996 the value specified in the structure pointed to by *value*, and if *ovalue* is not a null pointer, stores
16997 the previous value of the timer in the structure pointed to by *ovalue*.

16998 A timer value is defined by the **itimerval** structure, specified in **<sys/time.h>**. If *it_value* is non-
16999 zero, it shall indicate the time to the next timer expiration. If *it_interval* is non-zero, it shall
17000 specify a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 shall
17001 disable a timer, regardless of the value of *it_interval*. Setting *it_interval* to 0 shall disable a timer
17002 after its next expiration (assuming *it_value* is non-zero).

17003 Implementations may place limitations on the granularity of timer values. For each interval
17004 timer, if the requested timer value requires a finer granularity than the implementation supports,
17005 the actual timer value shall be rounded up to the next supported value.

17006 An XSI-conforming implementation provides each process with at least three interval timers,
17007 which are indicated by the *which* argument:

17008 ITIMER_REAL — Decrements in real time. A SIGALRM signal is delivered when this timer
17009 expires.

17010 ITIMER_VIRTUAL — Decrements in process virtual time. It runs only when the process is
17011 executing. A SIGVTALRM signal is delivered when it expires.

17012 ITIMER_PROF — Decrements both in process virtual time and when the system is running
17013 on behalf of the process. It is designed to be used by interpreters in
17014 statistically profiling the execution of interpreted programs. Each time the
17015 ITIMER_PROF timer expires, the SIGPROF signal is delivered.

17016 The interaction between *setitimer*( ) and any of *alarm*( ), *sleep*( ), or *usleep*( ) is unspecified.

17017 **RETURN VALUE**

17018 Upon successful completion, *getitimer*( ) or *setitimer*( ) shall return 0; otherwise, –1 shall be
17019 returned and *errno* set to indicate the error.

17020 **ERRORS**

17021 The *setitimer*( ) function shall fail if:

17022 [EINVAL] — The *value* argument is not in canonical form. (In canonical form, the number of
17023 microseconds is a non-negative integer less than 1,000,000 and the number of
17024 seconds is a non-negative integer.)

17025 The *getitimer*( ) and *setitimer*( ) functions may fail if:

17026 [EINVAL] — The *which* argument is not recognized.

17027 **EXAMPLES**

17028 None.

17029 **APPLICATION USAGE**

17030 None.

17031 **RATIONALE**

17032 None.

17033 **FUTURE DIRECTIONS**

17034 None.

17035 **SEE ALSO**

17036 *alarm*( ), *sleep*( ), *timer_getoverrun*( ), *ualarm*( ), *usleep*( ), the Base Definitions volume of
17037 IEEE Std 1003.1-200x, **<signal.h>**, **<sys/time.h>**

17038 **CHANGE HISTORY**

17039 First released in Issue 4, Version 2.

17040 **Issue 5**

17041 Moved from X/OPEN UNIX extension to BASE.

17042 **Issue 6**

17043 The **restrict** keyword is added to the *setitimer*( ) prototype for alignment with the
17044 ISO/IEC 9899: 1999 standard.

17045 **NAME**

17046 getlogin, getlogin_r — get login name

17047 **SYNOPSIS**

17048 ```
#include <unistd.h>
```

17049 ```
char *getlogin(void);
```

17050 TSF ```
int getlogin_r(char *name, size_t namesize);
```

17051

17052 **DESCRIPTION**

17053 The *getlogin*( ) function shall return a pointer to a string containing the user name associated by
17054 the login activity with the controlling terminal of the current process. If *getlogin*( ) returns a non-
17055 null pointer, then that pointer points to the name that the user logged in under, even if there are
17056 several login names with the same user ID.

17057 The *getlogin*( ) function need not be reentrant. A function that is not required to be reentrant is
17058 not required to be thread-safe.

17059 TSF The *getlogin_r*( ) function shall put the name associated by the login activity with the controlling
17060 terminal of the current process in the character array pointed to by *name*. The array is *namesize*
17061 characters long and should have space for the name and the terminating null character. The
17062 maximum size of the login name is {LOGIN_NAME_MAX}.

17063 If *getlogin_r*( ) is successful, *name* points to the name the user used at login, even if there are
17064 several login names with the same user ID.

17065 **RETURN VALUE**

17066 Upon successful completion, *getlogin*( ) shall return a pointer to the login name or a null pointer
17067 if the user's login name cannot be found. Otherwise, it shall return a null pointer and set *errno* to
17068 indicate the error.

17069 The return value from *getlogin*( ) may point to static data whose content is overwritten by each
17070 call.

17071 TSF If successful, the *getlogin_r*( ) function shall return zero; otherwise, an error number shall be
17072 returned to indicate the error.

17073 **ERRORS**

17074 The *getlogin*( ) and *getlogin_r*( ) functions may fail if:

17075 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

17076 [ENFILE] The maximum allowable number of files is currently open in the system.

17077 [ENXIO] The calling process has no controlling terminal.

17078 The *getlogin_r*( ) function may fail if:

17079 TSF [ERANGE] The value of *namesize* is smaller than the length of the string to be returned
17080 including the terminating null character.
```

EXAMPLES

Getting the User Login Name

The following example calls the *getlogin*() function to obtain the name of the user associated with the calling process, and passes this information to the *getpwnam*() function to get the associated user database information.

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
...
char *lgn;
struct passwd *pw;
...
if ((lgn = getlogin()) == NULL || (pw = getpwnam(lgn)) == NULL) {
    fprintf(stderr, "Get of user information failed.\n"); exit(1);
    }
```

APPLICATION USAGE

Three names associated with the current process can be determined: *getpwuid*(*geteuid*()) shall return the name associated with the effective user ID of the process; *getlogin*() shall return the name associated with the current login activity; and *getpwuid*(*getuid*()) shall return the name associated with the real user ID of the process.

The *getlogin_r*() function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

The *getlogin*() function returns a pointer to the user's login name. The same user ID may be shared by several login names. If it is desired to get the user database entry that is used during login, the result of *getlogin*() should be used to provide the argument to the *getpwnam*() function. (This might be used to determine the user's login shell, particularly where a single user has multiple login shells with distinct login names, but the same user ID.)

The information provided by the *cuserid*() function, which was originally defined in the POSIX.1-1988 standard and subsequently removed, can be obtained by the following:

```
getpwuid(geteuid())
```

while the information provided by historical implementations of *cuserid*() can be obtained by:

```
getpwuid(getuid())
```

The thread-safe version of this function places the user name in a user-supplied buffer and returns a non-zero value if it fails. The non-thread-safe version may return the name in a static data area that may be overwritten by each call.

FUTURE DIRECTIONS

None.

SEE ALSO

getpwnam(), *getpwuid*(), *geteuid*(), *getuid*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<unistd.h>**

17123 **CHANGE HISTORY**

17124 First released in Issue 1. Derived from System V Release 2.0.

17125 **Issue 5**

17126 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
17127 VALUE section.

17128 The *getlogin_r*( ) function is included for alignment with the POSIX Threads Extension.

17129 A note indicating that the *getlogin*( ) function need not be reentrant is added to the
17130 DESCRIPTION.

17131 **Issue 6**

17132 The *getlogin_r*( ) function is marked as part of the Thread-Safe Functions option.

17133 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

17134 The following new requirements on POSIX implementations derive from alignment with the
17135 Single UNIX Specification:

17136 - In the RETURN VALUE section, the requirement to set *errno* on error is added.

17137 - The [EMFILE], [ENFILE], and [ENXIO] optional error conditions are added.

17138 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
17139 its avoidance of possibly using a static data area.

17140 **NAME**

17141 getmsg, getpmsg — receive next message from a STREAMS file (**STREAMS**)

17142 **SYNOPSIS**

17143 XSR
```
#include <stropts.h>

int getmsg(int fildes, struct strbuf *restrict ctlptr,
    struct strbuf *restrict dataptr, int *restrict flagsp);
int getpmsg(int fildes, struct strbuf *restrict ctlptr,
    struct strbuf *restrict dataptr, int *restrict bandp,
    int *restrict flagsp);
```
17149

17150 **DESCRIPTION**

17151 The *getmsg*( ) function shall retrieve the contents of a message located at the head of the
17152 STREAM head read queue associated with a STREAMS file and place the contents into one or
17153 more buffers. The message contains either a data part, a control part, or both. The data and
17154 control parts of the message shall be placed into separate buffers, as described below. The |
17155 semantics of each part are defined by the originator of the message.

17156 The *getpmsg*( ) function shall be equivalent to *getmsg*( ), except that it provides finer control over |
17157 the priority of the messages received. Except where noted, all requirements on *getmsg*( ) also |
17158 pertain to *getpmsg*( ).

17159 The *fildes* argument specifies a file descriptor referencing a STREAMS-based file.

17160 The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure, in which the *buf* member points
17161 to a buffer in which the data or control information is to be placed, and the *maxlen* member
17162 indicates the maximum number of bytes this buffer can hold. On return, the *len* member shall |
17163 contain the number of bytes of data or control information actually received. The *len* member |
17164 shall be set to 0 if there is a zero-length control or data part and *len* shall be set to –1 if no data or |
17165 control information is present in the message. |

17166 When *getmsg*( ) is called, *flagsp* should point to an integer that indicates the type of message the
17167 process is able to receive. This is described further below.

17168 The *ctlptr* argument is used to hold the control part of the message, and *dataptr* is used to hold
17169 the data part of the message. If *ctlptr* (or *dataptr*) is a null pointer or the *maxlen* member is –1, the |
17170 control (or data) part of the message shall not be processed and shall be left on the STREAM |
17171 head read queue, and if the *ctlptr* (or *dataptr*) is not a null pointer, *len* shall be set to –1. If the |
17172 *maxlen* member is set to 0 and there is a zero-length control (or data) part, that zero-length part |
17173 shall be removed from the read queue and *len* shall be set to 0. If the *maxlen* member is set to 0 |
17174 and there are more than 0 bytes of control (or data) information, that information shall be left on |
17175 the read queue and *len* shall be set to 0. If the *maxlen* member in *ctlptr* (or *dataptr*) is less than the |
17176 control (or data) part of the message, *maxlen* bytes shall be retrieved. In this case, the remainder |
17177 of the message shall be left on the STREAM head read queue and a non-zero return value shall |
17178 be provided. |

17179 By default, *getmsg*( ) shall process the first available message on the STREAM head read queue. |
17180 However, a process may choose to retrieve only high-priority messages by setting the integer |
17181 pointed to by *flagsp* to RS_HIPRI. In this case, *getmsg*( ) shall only process the next message if it is |
17182 a high-priority message. When the integer pointed to by *flagsp* is 0, any available message shall |
17183 be retrieved. In this case, on return, the integer pointed to by *flagsp* shall be set to RS_HIPRI if a |
17184 high-priority message was retrieved, or 0 otherwise. |

17185 For *getpmsg*( ), the flags are different. The *flagsp* argument points to a bitmask with the following
17186 mutually-exclusive flags defined: MSG_HIPRI, MSG_BAND, and MSG_ANY. Like *getmsg*( ),

getpmsg( ) shall process the first available message on the STREAM head read queue. A process may choose to retrieve only high-priority messages by setting the integer pointed to by *flagsp* to MSG_HIPRI and the integer pointed to by *bandp* to 0. In this case, *getpmsg*( ) shall only process the next message if it is a high-priority message. In a similar manner, a process may choose to retrieve a message from a particular priority band by setting the integer pointed to by *flagsp* to MSG_BAND and the integer pointed to by *bandp* to the priority band of interest. In this case, *getpmsg*( ) shall only process the next message if it is in a priority band equal to, or greater than, the integer pointed to by *bandp*, or if it is a high-priority message. If a process wants to get the first message off the queue, the integer pointed to by *flagsp* should be set to MSG_ANY and the integer pointed to by *bandp* should be set to 0. On return, if the message retrieved was a high-priority message, the integer pointed to by *flagsp* shall be set to MSG_HIPRI and the integer pointed to by *bandp* shall be set to 0. Otherwise, the integer pointed to by *flagsp* shall be set to MSG_BAND and the integer pointed to by *bandp* shall be set to the priority band of the message.

If O_NONBLOCK is not set, *getmsg*( ) and *getpmsg*( ) shall block until a message of the type specified by *flagsp* is available at the front of the STREAM head read queue. If O_NONBLOCK is set and a message of the specified type is not present at the front of the read queue, *getmsg*( ) and *getpmsg*( ) shall fail and set *errno* to [EAGAIN].

If a hangup occurs on the STREAM from which messages are retrieved, *getmsg*( ) and *getpmsg*( ) shall continue to operate normally, as described above, until the STREAM head read queue is empty. Thereafter, they shall return 0 in the *len* members of *ctlptr* and *dataptr*.

RETURN VALUE

Upon successful completion, *getmsg*( ) and *getpmsg*( ) shall return a non-negative value. A value of 0 indicates that a full message was read successfully. A return value of MORECTL indicates that more control information is waiting for retrieval. A return value of MOREDATA indicates that more data is waiting for retrieval. A return value of the bitwise-logical OR of MORECTL and MOREDATA indicates that both types of information remain. Subsequent *getmsg*( ) and *getpmsg*( ) calls shall retrieve the remainder of the message. However, if a message of higher priority has come in on the STREAM head read queue, the next call to *getmsg*( ) or *getpmsg*( ) shall retrieve that higher-priority message before retrieving the remainder of the previous message.

If the high priority control part of the message is consumed, the message shall be placed back on the queue as a normal message of band 0. Subsequent *getmsg*( ) and *getpmsg*( ) calls shall retrieve the remainder of the message. If, however, a priority message arrives or already exists on the STREAM head, the subsequent call to *getmsg*( ) or *getpmsg*( ) shall retrieve the higher-priority message before retrieving the remainder of the message that was put back.

Upon failure, *getmsg*( ) and *getpmsg*( ) shall return –1 and set *errno* to indicate the error.

ERRORS

The *getmsg*( ) and *getpmsg*( ) functions shall fail if:

[EAGAIN] The O_NONBLOCK flag is set and no messages are available.

[EBADF] The *fildes* argument is not a valid file descriptor open for reading.

[EBADMSG] The queued message to be read is not valid for *getmsg*( ) or *getpmsg*( ) or a pending file descriptor is at the STREAM head.

[EINTR] A signal was caught during *getmsg*( ) or *getpmsg*( ).

[EINVAL] An illegal value was specified by *flagsp*, or the STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

[ENOSTR] A STREAM is not associated with *fildes*.

In addition, *getmsg*() and *getpmsg*() shall fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of *errno* does not reflect the result of *getmsg*() or *getpmsg*() but reflects the prior error.

EXAMPLES

Getting Any Message

In the following example, the value of *fd* is assumed to refer to an open STREAMS file. The call to *getmsg*() retrieves any available message on the associated STREAM-head read queue, returning control and data information to the buffers pointed to by *ctrlbuf* and *databuf*, respectively.

```
#include <stropts.h>
...
int fd;
char ctrlbuf[128];
char databuf[512];
struct strbuf ctrl;
struct strbuf data;
int flags = 0;
int ret;

ctrl.buf = ctrlbuf;
ctrl.maxlen = sizeof(ctrlbuf);

data.buf = databuf;
data.maxlen = sizeof(databuf);

ret = getmsg (fd, &ctrl, &data, &flags);
```

Getting the First Message off the Queue

In the following example, the call to *getpmsg*() retrieves the first available message on the associated STREAM-head read queue.

```
#include <stropts.h>
...

int fd;
char ctrlbuf[128];
char databuf[512];
struct strbuf ctrl;
struct strbuf data;
int band = 0;
int flags = MSG_ANY;
int ret;

ctrl.buf = ctrlbuf;
ctrl.maxlen = sizeof(ctrlbuf);

data.buf = databuf;
data.maxlen = sizeof(databuf);

ret = getpmsg (fd, &ctrl, &data, &band, &flags);
```

17275 **APPLICATION USAGE**

17276 None.

17277 **RATIONALE**

17278 None.

17279 **FUTURE DIRECTIONS**

17280 None.

17281 **SEE ALSO**

17282 *poll*( ), *putmsg*( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
17283 **<stropts.h>**, Section 2.6 (on page 488)

17284 **CHANGE HISTORY**

17285 First released in Issue 4, Version 2.

17286 **Issue 5**

17287 Moved from X/OPEN UNIX extension to BASE.

17288 A paragraph regarding ‘‘high-priority control parts of messages’’ is added to the RETURN
17289 VALUE section.

17290 **Issue 6**

17291 This function is marked as part of the XSI STREAMS Option Group.

17292 The **restrict** keyword is added to the *getmsg*( ) and *getpmsg*( ) prototypes for alignment with the
17293 ISO/IEC 9899: 1999 standard.

17294 **NAME**

17295 getnameinfo — get name information

17296 **SYNOPSIS**

17297 `#include <sys/socket.h>`
17298 `#include <netdb.h>`

17299 `int getnameinfo(const struct sockaddr *restrict sa, socklen_t salen,`
17300 `char *restrict node, socklen_t nodelen, char *restrict service,`
17301 `socklen_t servicelen, unsigned flags);`

17302 **DESCRIPTION**

17303 The *getnameinfo*( ) function shall translate a socket address to a node name and service location,
17304 all of which are defined as in *getaddrinfo*( ).

17305 The *sa* argument points to a socket address structure to be translated.

17306 If the *node* argument is non-NULL and the *nodelen* argument is non-zero, then the *node* argument
17307 points to a buffer able to contain up to *nodelen* characters that receives the node name as a null-
17308 terminated string. If the *node* argument is NULL or the *nodelen* argument is zero, the node name
17309 shall not be returned. If the node's name cannot be located, the numeric form of the node's
17310 address is returned instead of its name.

17311 If the *service* argument is non-NULL and the *servicelen* argument is non-zero, then the *service*
17312 argument points to a buffer able to contain up to *servicelen* bytes that receives the service name
17313 as a null-terminated string. If the *service* argument is NULL or the *servicelen* argument is zero,
17314 the service name shall not be returned. If the service's name cannot be located, the numeric form
17315 of the service address (for example, its port number) shall be returned instead of its name.

17316 The *flags* argument is a flag that changes the default actions of the function. By default the fully-
17317 qualified domain name (FQDN) for the host shall be returned, but:

17318 • If the flag bit NI_NOFQDN is set, only the node name portion of the FQDN shall be returned
17319 for local hosts.

17320 • If the flag bit NI_NUMERICHOST is set, the numeric form of the host's address shall be
17321 returned instead of its name, under all circumstances.

17322 • If the flag bit NI_NAMEREQD is set, an error shall be returned if the host's name cannot be
17323 located.

17324 • If the flag bit NI_NUMERICSERV is set, the numeric form of the service address shall be
17325 returned (for example, its port number) instead of its name, under all circumstances.

17326 • If the flag bit NI_DGRAM is set, this indicates that the service is a datagram service
17327 (SOCK_DGRAM). The default behavior shall assume that the service is a stream service
17328 (SOCK_STREAM).

17329 **Notes:**

17330 1. The two NI_NUMERICxxx flags are required to support the **–n** flag that many
17331 commands provide.

17332 2. The NI_DGRAM flag is required for the few AF_INET and AF_INET6 port numbers (for
17333 example, [512,514]) that represent different services for UDP and TCP.

17334 The *getnameinfo*( ) function shall be thread-safe.

17335 **RETURN VALUE**

17336 A zero return value for *getnameinfo*( ) indicates successful completion; a non-zero return value
17337 indicates failure. The possible values for the failures are listed in the ERRORS section.

17338 Upon successful completion, *getnameinfo*( ) shall return the *node* and *service* names, if requested,
17339 in the buffers provided. The returned names are always null-terminated strings.

17340 **ERRORS**

17341 The *getnameinfo*( ) function shall fail and return the corresponding value if:

17342 [EAI_AGAIN] The name could not be resolved at this time. Future attempts may succeed.

17343 [EAI_BADFLAGS]
17344 The *flags* had an invalid value.

17345 [EAI_FAIL] A non-recoverable error occurred.

17346 [EAI_FAMILY] The address family was not recognized or the address length was invalid for
17347 the specified family.

17348 [EAI_MEMORY] There was a memory allocation failure.

17349 [EAI_NONAME] The name does not resolve for the supplied parameters.

17350 NI_NAMEREQD is set and the host's name cannot be located, or both
17351 *nodename* and *servname* were null.

17352 [EAI_SYSTEM] A system error occurred. The error code can be found in *errno*.

17353 **EXAMPLES**

17354 None.

17355 **APPLICATION USAGE**

17356 If the returned values are to be used as part of any further name resolution (for example, passed
17357 to *getaddrinfo*( )), applications should provide buffers large enough to store any result possible
17358 on the system.

17359 **RATIONALE**

17360 None.

17361 **FUTURE DIRECTIONS**

17362 None.

17363 **SEE ALSO**

17364 *gai_strerror*( ), *getaddrinfo*( ), *getservbyname*( ), *getservbyport*( ), *inet_ntop*( ), *socket*( ), the Base
17365 Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**, **<sys/socket.h>**

17366 **CHANGE HISTORY**

17367 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

17368 The **restrict** keyword is added to the *getnameinfo*( ) prototype for alignment with the
17369 ISO / IEC 9899: 1999 standard.

17370 **NAME**

17371 getnetbyaddr — network database functions

17372 **SYNOPSIS**

17373 `#include <netdb.h>`

17374 `struct netent *getnetbyaddr(uint32_t net, int type);`

17375 **DESCRIPTION**

17376 Refer to *endnetent*( ).

17377 **NAME**

17378 getnetbyname — network database functions

17379 **SYNOPSIS**

17380 `#include <netdb.h>`

17381 `struct netent *getnetbyname(const char *`*`name`*`);`

17382 **DESCRIPTION**

17383 Refer to *endnetent*( ).

17384 **NAME**

17385 getnetent — network database functions

17386 **SYNOPSIS**

17387 `#include <netdb.h>`

17388 `struct netent *getnetent(void);`

17389 **DESCRIPTION**

17390 Refer to *endnetent*( ).

17391 **NAME**

17392 getopt, optarg, opterr, optind, optopt — command option parsing

17393 **SYNOPSIS**

17394 ```
#include <unistd.h>
```

17395 ```
int getopt(int argc, char * const argv[], const char *optstring);
```
17396 ```
extern char *optarg;
```
17397 ```
extern int optind, opterr, optopt;
```

17398 **DESCRIPTION**

17399 The *getopt*( ) function is a command-line parser that shall follow Utility Syntax Guidelines 3, 4, 5, |
17400 6, 7, 9, and 10 in the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax |
17401 Guidelines. |

17402 The parameters *argc* and *argv* are the argument count and argument array as passed to *main*( )
17403 (see *exec*). The argument *optstring* is a string of recognized option characters; if a character is
17404 followed by a colon, the option takes an argument. All option characters allowed by Utility
17405 Syntax Guideline 3 are allowed in *optstring*. The implementation may accept other characters as
17406 an extension.

17407 The variable *optind* is the index of the next element of the *argv*[] vector to be processed. It shall |
17408 be initialized to 1 by the system, and *getopt*( ) shall update it when it finishes with each element |
17409 of *argv*[ ]. When an element of *argv*[ ] contains multiple option characters, it is unspecified how |
17410 *getopt*( ) determines which options have already been processed.

17411 The *getopt*( ) function shall return the next option character (if one is found) from *argv* that
17412 matches a character in *optstring*, if there is one that matches. If the option takes an argument,
17413 *getopt*( ) shall set the variable *optarg* to point to the option-argument as follows:

17414 1. If the option was the last character in the string pointed to by an element of *argv*, then
17415 *optarg* shall contain the next element of *argv*, and *optind* shall be incremented by 2. If the |
17416 resulting value of *optind* is greater than *argc*, this indicates a missing option-argument, and |
17417 *getopt*( ) shall return an error indication.

17418 2. Otherwise, *optarg* shall point to the string following the option character in that element of |
17419 *argv*, and *optind* shall be incremented by 1. |

17420 If, when *getopt*( ) is called:

17421 `argv[optind]` is a null pointer
17422 `*argv[optind]` is not the character –
17423 `argv[optind]` points to the string `"–"`

17424 *getopt*( ) shall return –1 without changing *optind*. If:

17425 `argv[optind]` points to the string `"––"`

17426 *getopt*( ) shall return –1 after incrementing *optind*.

17427 If *getopt*( ) encounters an option character that is not contained in *optstring*, it shall return the
17428 question-mark (`'?'`) character. If it detects a missing option-argument, it shall return the colon
17429 character (`':'`) if the first character of *optstring* was a colon, or a question-mark character (`'?'`)
17430 otherwise. In either case, *getopt*( ) shall set the variable *optopt* to the option character that caused
17431 the error. If the application has not set the variable *opterr* to 0 and the first character of *optstring*
17432 is not a colon, *getopt*( ) shall also print a diagnostic message to *stderr* in the format specified for
17433 the *getopts* utility.

17434 The *getopt*( ) function need not be reentrant. A function that is not required to be reentrant is not
17435 required to be thread-safe.

17436 **RETURN VALUE**

17437 The *getopt*( ) function shall return the next option character specified on the command line.

17438 A colon (':') shall be returned if *getopt*( ) detects a missing argument and the first character of
17439 *optstring* was a colon (':').

17440 A question mark ('?') shall be returned if *getopt*( ) encounters an option character not in
17441 *optstring* or detects a missing argument and the first character of *optstring* was not a colon (':').

17442 Otherwise, *getopt*( ) shall return –1 when all command line options are parsed.

17443 **ERRORS**

17444 No errors are defined.

17445 **EXAMPLES**

17446 **Parsing Command Line Options**

17447 The following code fragment shows how you might process the arguments for a utility that can
17448 take the mutually-exclusive options *a* and *b* and the options *f* and *o*, both of which require
17449 arguments:

```
17450 #include <unistd.h>

17451 int
17452 main(int argc, char *argv[ ])
17453 {
17454     int c;
17455     int bflg, aflg, errflg;
17456     char *ifile;
17457     char *ofile;
17458     extern char *optarg;
17459     extern int optind, optopt;
17460     . . .
17461     while ((c = getopt(argc, argv, ":abf:o:")) != -1) {
17462         switch(c) {
17463         case 'a':
17464             if (bflg)
17465                 errflg++;
17466             else
17467                 aflg++;
17468             break;
17469         case 'b':
17470             if (aflg)
17471                 errflg++;
17472             else {
17473                 bflg++;
17474                 bproc();
17475             }
17476             break;
17477         case 'f':
17478             ifile = optarg;
17479             break;
17480         case 'o':
17481             ofile = optarg;
17482             break;
```

```
                case ':':       /* -f or -o without operand */
                        fprintf(stderr,
                                "Option -%c requires an operand\n", optopt);
                        errflg++;
                        break;
            case '?':
                        fprintf(stderr,
                                "Unrecognized option: -%c\n", optopt);
                errflg++;
            }
        }
        if (errflg) {
            fprintf(stderr, "usage: . . . ");
            exit(2);
        }
        for ( ; optind < argc; optind++) {
            if (access(argv[optind], R_OK)) {
        . . .
    }
```

This code accepts any of the following as equivalent:

```
cmd –ao arg path path
cmd –a –o arg path path
cmd –o arg –a path path
cmd –a –o arg –– path path
cmd –a –oarg path path
cmd –aoarg path path
```

Checking Options and Arguments

The following example parses a set of command line options and prints messages to standard output for each option and argument that it encounters.

```
#include <unistd.h>
#include <stdio.h>
...
int c;
char *filename;
extern char *optarg;
extern int optind, optopt, opterr;
...
while ((c = getopt(argc, argv, ":abf:")) != -1) {
    switch(c) {
    case 'a':
        printf("a is set\n");
        break;
    case 'b':
        printf("b is set\n");
        break;
    case 'f':
        filename = optarg;
        printf("filename is %s\n", filename);
        break;
```

```
17532           case ':':
17533               printf("-%c without filename\n", optopt);
17534               break;
17535           case '?':
17536               printf("unknown arg %c\n", optopt);
17537               break;
17538           }
17539       }
```

17540 **Selecting Options from the Command Line**

17541 The following example selects the type of database routines the user wants to use based on the
17542 *Options* argument.

```
17543       #include <unistd.h>
17544       #include <string.h>
17545       ...
17546       char *Options = "hdbtl";
17547       ...
17548       int dbtype, i;
17549       char c;
17550       char *st;
17551       ...
17552       dbtype = 0;
17553       while ((c = getopt(argc, argv, Options)) != -1) {
17554           if ((st = strchr(Options, c)) != NULL) {
17555               dbtype = st - Options;
17556               break;
17557           }
17558       }
```

17559 **APPLICATION USAGE**

17560 The *getopt*( ) function is only required to support option characters included in Utility Syntax
17561 Guideline 3. Many historical implementations of *getopt*( ) support other characters as options.
17562 This is an allowed extension, but applications that use extensions are not maximally portable.
17563 Note that support for multi-byte option characters is only possible when such characters can be
17564 represented as type **int**.

17565 **RATIONALE**

17566 The *optopt* variable represents historical practice and allows the application to obtain the identity
17567 of the invalid option.

17568 The description has been written to make it clear that *getopt*( ), like the *getopts* utility, deals with
17569 option-arguments whether separated from the option by <blank>s or not. Note that the
17570 requirements on *getopt*( ) and *getopts* are more stringent than the Utility Syntax Guidelines.

17571 The *getopt*( ) function shall return –1, rather than EOF, so that **<stdio.h>** is not required.

17572 The special significance of a colon as the first character of *optstring* makes *getopt*( ) consistent
17573 with the *getopts* utility. It allows an application to make a distinction between a missing
17574 argument and an incorrect option letter without having to examine the option letter. It is true
17575 that a missing argument can only be detected in one case, but that is a case that has to be
17576 considered.

17577 **FUTURE DIRECTIONS**

17578 None.

17579 **SEE ALSO**

17580 *exec*, the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**, the Shell and Utilities
17581 volume of IEEE Std 1003.1-200x

17582 **CHANGE HISTORY**

17583 First released in Issue 1. Derived from Issue 1 of the SVID.

17584 **Issue 5**

17585 A note indicating that the *getopt*( ) function need not be reentrant is added to the DESCRIPTION.

17586 **Issue 6**

17587 IEEE PASC Interpretation 1003.2 #150 is applied.

17588 **NAME**

17589 getpeername — get the name of the peer socket

17590 **SYNOPSIS**

17591
```
#include <sys/socket.h>
```

17592
```
int getpeername(int socket, struct sockaddr *restrict address,
17593     socklen_t *restrict address_len);
```

17594 **DESCRIPTION**

17595 The *getpeername*( ) function shall retrieve the peer address of the specified socket, store this
17596 address in the **sockaddr** structure pointed to by the *address* argument, and store the length of this
17597 address in the object pointed to by the *address_len* argument.

17598 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
17599 the stored address shall be truncated.

17600 If the protocol permits connections by unbound clients, and the peer is not bound, then the value
17601 stored in the object pointed to by *address* is unspecified.

17602 **RETURN VALUE**

17603 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
17604 indicate the error.

17605 **ERRORS**

17606 The *getpeername*( ) function shall fail if:

17607 [EBADF] The *socket* argument is not a valid file descriptor.

17608 [EINVAL] The socket has been shut down.

17609 [ENOTCONN] The socket is not connected or otherwise has not had the peer prespecified.

17610 [ENOTSOCK] The *socket* argument does not refer to a socket.

17611 [EOPNOTSUPP] The operation is not supported for the socket protocol.

17612 The *getpeername*( ) function may fail if:

17613 [ENOBUFS] Insufficient resources were available in the system to complete the call.

17614 **EXAMPLES**

17615 None.

17616 **APPLICATION USAGE**

17617 None.

17618 **RATIONALE**

17619 None.

17620 **FUTURE DIRECTIONS**

17621 None.

17622 **SEE ALSO**

17623 *accept*( ), *bind*( ), *getsockname*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
17624 **<sys/socket.h>**

17625 **CHANGE HISTORY**

17626 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

17627 The **restrict** keyword is added to the *getpeername*( ) prototype for alignment with the
17628 ISO / IEC 9899: 1999 standard.

17629 **NAME**

17630 getpgid — get the process group ID for a process

17631 **SYNOPSIS**

17632 XSI
```
#include <unistd.h>

pid_t getpgid(pid_t pid);
```
17633

17634

17635 **DESCRIPTION**

17636 The *getpgid*( ) function shall return the process group ID of the process whose process ID is equal
17637 to *pid*. If *pid* is equal to 0, *getpgid*( ) shall return the process group ID of the calling process.

17638 **RETURN VALUE**

17639 Upon successful completion, *getpgid*( ) shall return a process group ID. Otherwise, it shall return
17640 (**pid_t**)–1 and set *errno* to indicate the error.

17641 **ERRORS**

17642 The *getpgid*( ) function shall fail if:

17643 [EPERM] The process whose process ID is equal to *pid* is not in the same session as the
17644 calling process, and the implementation does not allow access to the process
17645 group ID of that process from the calling process.

17646 [ESRCH] There is no process with a process ID equal to *pid*.

17647 The *getpgid*( ) function may fail if:

17648 [EINVAL] The value of the *pid* argument is invalid.

17649 **EXAMPLES**

17650 None.

17651 **APPLICATION USAGE**

17652 None.

17653 **RATIONALE**

17654 None.

17655 **FUTURE DIRECTIONS**

17656 None.

17657 **SEE ALSO**

17658 *exec*, *fork*( ), *getpgrp*( ), *getpid*( ), *getsid*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of
17659 IEEE Std 1003.1-200x, **<unistd.h>**

17660 **CHANGE HISTORY**

17661 First released in Issue 4, Version 2.

17662 **Issue 5**

17663 Moved from X/OPEN UNIX extension to BASE.

17664 **NAME**

17665 getpgrp — get the process group ID of the calling process

17666 **SYNOPSIS**

17667
```
#include <unistd.h>
```

17668
```
pid_t getpgrp(void);
```

17669 **DESCRIPTION**

17670 The *getpgrp*( ) function shall return the process group ID of the calling process.

17671 **RETURN VALUE**

17672 The *getpgrp*( ) function shall always be successful and no return value is reserved to indicate an |
17673 error. |

17674 **ERRORS**

17675 No errors are defined.

17676 **EXAMPLES**

17677 None.

17678 **APPLICATION USAGE**

17679 None.

17680 **RATIONALE**

17681 4.3 BSD provides a *getpgrp*( ) function that returns the process group ID for a specified process. |
17682 Although this function supports job control, all known job control shells always specify the |
17683 calling process with this function. Thus, the simpler System V *getpgrp*( ) suffices, and the added
17684 complexity of the 4.3 BSD *getpgrp*( ) is provided by the XSI extension *getpgid*( ).

17685 **FUTURE DIRECTIONS**

17686 None.

17687 **SEE ALSO**

17688 *exec*, *fork*( ), *getpgid*( ), *getpid*( ), *getppid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of
17689 IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

17690 **CHANGE HISTORY**

17691 First released in Issue 1. Derived from Issue 1 of the SVID.

17692 **Issue 6**

17693 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

17694 The following new requirements on POSIX implementations derive from alignment with the
17695 Single UNIX Specification:

17696 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
17697 required for conforming implementations of previous POSIX specifications, it was not
17698 required for UNIX applications.

17699 **NAME**

17700 getpid — get the process ID

17701 **SYNOPSIS**

17702
```
#include <unistd.h>
```

17703
```
pid_t getpid(void);
```

17704 **DESCRIPTION**

17705 The *getpid*( ) function shall return the process ID of the calling process.

17706 **RETURN VALUE**

17707 The *getpid*( ) function shall always be successful and no return value is reserved to indicate an |
17708 error.

17709 **ERRORS**

17710 No errors are defined.

17711 **EXAMPLES**

17712 None.

17713 **APPLICATION USAGE**

17714 None.

17715 **RATIONALE**

17716 None.

17717 **FUTURE DIRECTIONS**

17718 None.

17719 **SEE ALSO**

17720 *exec*, *fork*( ), *getpgrp*( ), *getppid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of
17721 IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

17722 **CHANGE HISTORY**

17723 First released in Issue 1. Derived from Issue 1 of the SVID.

17724 **Issue 6**

17725 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

17726 The following new requirements on POSIX implementations derive from alignment with the
17727 Single UNIX Specification:

17728 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
17729 required for conforming implementations of previous POSIX specifications, it was not
17730 required for UNIX applications.

17731 **NAME**

17732 getpmsg — receive next message from a STREAMS file

17733 **SYNOPSIS**

17734 XSI
```
#include <stropts.h>

int getpmsg(int fildes, struct strbuf *restrict ctlptr,
    struct strbuf *restrict dataptr, int *restrict bandp,
    int *restrict flagsp);
```
17735
17736
17737
17738

17739 **DESCRIPTION**

17740 Refer to *getmsg*( ).

17741 **NAME**

17742 getppid — get the parent process ID

17743 **SYNOPSIS**

17744 `#include <unistd.h>`

17745 `pid_t getppid(void);`

17746 **DESCRIPTION**

17747 The *getppid*( ) function shall return the parent process ID of the calling process.

17748 **RETURN VALUE**

17749 The *getppid*( ) function shall always be successful and no return value is reserved to indicate an |
17750 error.

17751 **ERRORS**

17752 No errors are defined.

17753 **EXAMPLES**

17754 None.

17755 **APPLICATION USAGE**

17756 None.

17757 **RATIONALE**

17758 None.

17759 **FUTURE DIRECTIONS**

17760 None.

17761 **SEE ALSO**

17762 *exec*, *fork*( ), *getpgid*( ), *getpgrp*( ), *getpid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of
17763 IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

17764 **CHANGE HISTORY**

17765 First released in Issue 1. Derived from Issue 1 of the SVID.

17766 **Issue 6**

17767 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

17768 The following new requirements on POSIX implementations derive from alignment with the
17769 Single UNIX Specification:

17770 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
17771 required for conforming implementations of previous POSIX specifications, it was not
17772 required for UNIX applications.

17773 **NAME**

17774 getpriority, setpriority — get and set the nice value

17775 **SYNOPSIS**

17776 XSI
```
#include <sys/resource.h>

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int value);
```
17779

17780 **DESCRIPTION**

17781 The *getpriority*( ) function shall obtain the nice value of a process, process group, or user. The
17782 *setpriority*( ) function shall set the nice value of a process, process group, or user to
17783 *value*+{NZERO}.

17784 Target processes are specified by the values of the *which* and *who* arguments. The *which*
17785 argument may be one of the following values: PRIO_PROCESS, PRIO_PGRP, or PRIO_USER,
17786 indicating that the *who* argument is to be interpreted as a process ID, a process group ID, or an
17787 effective user ID, respectively. A 0 value for the *who* argument specifies the current process,
17788 process group, or user.

17789 The nice value set with *setpriority*( ) shall be applied to the process. If the process is multi-
17790 threaded, the nice value shall affect all system scope threads in the process.

17791 If more than one process is specified, *getpriority*( ) shall return value {NZERO} less than the
17792 lowest nice value pertaining to any of the specified processes, and *setpriority*( ) shall set the nice
17793 values of all of the specified processes to *value*+{NZERO}.

17794 The default nice value is {NZERO}; lower nice values shall cause more favorable scheduling.
17795 While the range of valid nice values is [0,{NZERO}*2–1], implementations may enforce more
17796 restrictive limits. If *value*+{NZERO} is less than the system's lowest supported nice value,
17797 *setpriority*( ) shalls et the nice value to the lowest supported value; if *value*+{NZERO} is greater
17798 than the system's highest supported nice value, *setpriority*( ) shall set the nice value to the highest
17799 supported value.

17800 Only a process with appropriate privileges can lower its nice value.

17801 PS | TPS Any processes or threads using SCHED_FIFO or SCHED_RR shall be unaffected by a call to
17802 *setpriority*( ). This is not considered an error. A process which subsequently reverts to
17803 SCHED_OTHER need not have its priority affected by such a *setpriority*( ) call.

17804 The effect of changing the nice value may vary depending on the process-scheduling algorithm
17805 in effect.

17806 Since *getpriority*( ) can return the value –1 on successful completion, it is necessary to set *errno* to
17807 0 prior to a call to *getpriority*( ). If *getpriority*( ) returns the value –1, then *errno* can be checked to
17808 see if an error occurred or if the value is a legitimate nice value.

17809 **RETURN VALUE**

17810 Upon successful completion, *getpriority*( ) shall return an integer in the range from –{NZERO} to
17811 {NZERO}–1. Otherwise, –1 shall be returned and *errno* set to indicate the error.

17812 Upon successful completion, *setpriority*( ) shall return 0; otherwise, –1 shall be returned and *errno*
17813 set to indicate the error.

17814 **ERRORS**

17815 The *getpriority*( ) and *setpriority*( ) functions shall fail if:

17816 [ESRCH] No process could be located using the *which* and *who* argument values
17817 specified.

17818 [EINVAL] The value of the *which* argument was not recognized, or the value of the *who*
17819 argument is not a valid process ID, process group ID, or user ID.

17820 In addition, *setpriority*( ) may fail if:

17821 [EPERM] A process was located, but neither the real nor effective user ID of the
17822 executing process match the effective user ID of the process whose nice value
17823 is being changed.

17824 [EACCES] A request was made to change the nice value to a lower numeric value and
17825 the current process does not have appropriate privileges.

17826 **EXAMPLES**

17827 **Using getpriority( )**

17828 The following example returns the current scheduling priority for the process ID returned by the
17829 call to *getpid*( ).

```
17830 #include <sys/resource.h>
17831 ...
17832 int which = PRIO_PROCESS;
17833 id_t pid;
17834 int ret;

17835 pid = getpid();
17836 ret = getpriority(which, pid);
```

17837 **Using setpriority( )**

17838 The following example sets the priority for the current process ID to –20.

```
17839 #include <sys/resource.h>
17840 ...
17841 int which = PRIO_PROCESS;
17842 id_t pid;
17843 int priority = -20;
17844 int ret;

17845 pid = getpid();
17846 ret = setpriority(which, pid, priority);
```

17847 **APPLICATION USAGE**

17848 The *getpriority*( ) and *setpriority*( ) functions work with an offset nice value (nice value
17849 –{NZERO}). The nice value is in the range [0,2*{NZERO} –1], while the return value for
17850 *getpriority*( ) and the third parameter for *setpriority*( ) are in the range [–{NZERO},{NZERO} –1].

17851 **RATIONALE**

17852 None.

17853 **FUTURE DIRECTIONS**

17854 None.

17855 **SEE ALSO**

17856 *nice*( ), *sched_get_priority_max*( ), *sched_setscheduler*( ), the Base Definitions volume of
17857 IEEE Std 1003.1-200x, **<sys/resource.h>**

17858 **CHANGE HISTORY**

17859 First released in Issue 4, Version 2.

17860 **Issue 5**

17861 Moved from X/OPEN UNIX extension to BASE.

17862 The DESCRIPTION is reworded in terms of the nice value rather than *priority* to avoid confusion
17863 with functionality in the POSIX Realtime Extension.

17864 **NAME**

17865 getprotobyname — network protocol database functions

17866 **SYNOPSIS**

17867 `#include <netdb.h>`

17868 `struct protoent *getprotobyname(const char *`*`name`*`);`

17869 **DESCRIPTION**

17870 Refer to *endprotoent*( ).

17871 **NAME**

17872 getprotobynumber — network protocol database functions

17873 **SYNOPSIS**

17874 `#include <netdb.h>`

17875 `struct protoent *getprotobynumber(int proto);`

17876 **DESCRIPTION**

17877 Refer to *endprotoent*( ).

17878 **NAME**

17879 getprotoent — network protocol database functions

17880 **SYNOPSIS**

17881 `#include <netdb.h>`

17882 `struct protoent *getprotoent(void);`

17883 **DESCRIPTION**

17884 Refer to *endprotoent*( ).

17885 **NAME**

17886 getpwent — get user database entry

17887 **SYNOPSIS**

17888 XSI
```
#include <pwd.h>

struct passwd *getpwent(void);
```
17889

17890

17891 **DESCRIPTION**

17892 Refer to *endpwent*( ).

17893 **NAME**

17894 getpwnam, getpwnam_r — search user database for a name

17895 **SYNOPSIS**

17896 ```
#include <pwd.h>
```

17897 ```
struct passwd *getpwnam(const char *name);
```

17898 TSF ```
int getpwnam_r(const char *name, struct passwd *pwd, char *buffer,
    size_t bufsize, struct passwd **result);
```

17900

17901 **DESCRIPTION**

17902 The *getpwnam*( ) function shall search the user database for an entry with a matching *name*.

17903 The *getpwnam*( ) function need not be reentrant. A function that is not required to be reentrant is
17904 not required to be thread-safe.

17905 Applications wishing to check for error situations should set *errno* to 0 before calling
17906 *getpwnam*( ). If *getpwnam*( ) returns a null pointer and *errno* is non-zero, an error occurred.

17907 TSF The *getpwnam_r*( ) function shall update the **passwd** structure pointed to by *pwd* and store a
17908 pointer to that structure at the location pointed to by *result*. The structure shall contain an entry
17909 from the user database with a matching *name*. Storage referenced by the structure is allocated
17910 from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
17911 maximum size needed for this buffer can be determined with the {_SC_GETPW_R_SIZE_MAX}
17912 *sysconf*( ) parameter. A NULL pointer shall be returned at the location pointed to by *result* on
17913 error or if the requested entry is not found.

17914 **RETURN VALUE**

17915 The *getpwnam*( ) function shall return a pointer to a **struct passwd** with the structure as defined
17916 in **<pwd.h>** with a matching entry if found. A null pointer shall be returned if the requested
17917 entry is not found, or an error occurs. On error, *errno* shall be set to indicate the error.

17918 The return value may point to a static area which is overwritten by a subsequent call to
17919 *getpwent*( ), *getpwnam*( ), or *getpwuid*( ).

17920 TSF If successful, the *getpwnam_r*( ) function shall return zero; otherwise, an error number shall be
17921 returned to indicate the error.

17922 **ERRORS**

17923 The *getpwnam*( ) and *getpwnam_r*( ) functions may fail if:

17924 [EIO] An I/O error has occurred.

17925 [EINTR] A signal was caught during *getpwnam*( ).

17926 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

17927 [ENFILE] The maximum allowable number of files is currently open in the system.

17928 The *getpwnam_r*( ) function may fail if:

17929 TSF [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to
17930 be referenced by the resulting **passwd** structure.
```

EXAMPLES

Getting an Entry for the Login Name

The following example uses the *getlogin*( ) function to return the name of the user who logged in; this information is passed to the *getpwnam*( ) function to get the user database entry for that user.

```
#include <sys/types.h>
#include <pwd.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
...
char *lgn;
struct passwd *pw;
...
if ((lgn = getlogin()) == NULL || (pw = getpwnam(lgn)) == NULL) {
    fprintf(stderr, "Get of user information failed.\n"); exit(1);
}
...
```

APPLICATION USAGE

Three names associated with the current process can be determined: *getpwuid*(*geteuid*( )) returns the name associated with the effective user ID of the process; *getlogin*( ) returns the name associated with the current login activity; and *getpwuid*(*getuid*( )) returns the name associated with the real user ID of the process.

The *getpwnam_r*( ) function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getpwuid( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<pwd.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The *getpwnam_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *getpwnam*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *getpwnam_r*( ) function is marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION describing matching the *name*.

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EMFILE], [ENFILE], and [ENXIO] optional error conditions are added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from *bufsize* characters to bytes.

17987 **NAME**

17988 getpwuid, getpwuid_r — search user database for a user ID

17989 **SYNOPSIS**

17990 ```
#include <pwd.h>
```

17991 ```
struct passwd *getpwuid(uid_t uid);
```

17992 TSF ```
int getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,
17993   size_t bufsize, struct passwd **result);
```

17994

17995 **DESCRIPTION**

17996 The *getpwuid*( ) function shall search the user database for an entry with a matching *uid*.

17997 The *getpwuid*( ) function need not be reentrant. A function that is not required to be reentrant is
17998 not required to be thread-safe.

17999 Applications wishing to check for error situations should set *errno* to 0 before calling *getpwuid*( ).
18000 If *getpwuid*( ) returns a null pointer and *errno* is set to non-zero, an error occurred.

18001 TSF The *getpwuid_r*( ) function shall update the **passwd** structure pointed to by *pwd* and store a
18002 pointer to that structure at the location pointed to by *result*. The structure shall contain an entry
18003 from the user database with a matching *uid*. Storage referenced by the structure is allocated
18004 from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
18005 maximum size needed for this buffer can be determined with the {_SC_GETPW_R_SIZE_MAX}
18006 *sysconf*( ) parameter. A NULL pointer shall be returned at the location pointed to by *result* on
18007 error or if the requested entry is not found.

18008 **RETURN VALUE**

18009 The *getpwuid*( ) function shall return a pointer to a **struct passwd** with the structure as defined in
18010 **<pwd.h>** with a matching entry if found. A null pointer shall be returned if the requested entry
18011 is not found, or an error occurs. On error, *errno* shall be set to indicate the error.

18012 The return value may point to a static area which is overwritten by a subsequent call to
18013 *getpwent*( ), *getpwnam*( ), or *getpwuid*( ).

18014 TSF If successful, the *getpwuid_r*( ) function shall return zero; otherwise, an error number shall be
18015 returned to indicate the error.

18016 **ERRORS**

18017 The *getpwuid*( ) and *getpwuid_r*( ) functions may fail if:

18018 [EIO] An I/O error has occurred.

18019 [EINTR] A signal was caught during *getpwuid*( ).

18020 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

18021 [ENFILE] The maximum allowable number of files is currently open in the system.

18022 The *getpwuid_r*( ) function may fail if:

18023 TSF [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to
18024 be referenced by the resulting **passwd** structure.
```

EXAMPLES

Getting an Entry for the Root User

The following example gets the user database entry for the user with user ID 0 (root).

```
#include <sys/types.h>
#include <pwd.h>
...
uid_t id = 0;
struct passwd *pwd;

pwd = getpwuid(id);
```

Finding the Name for the Effective User ID

The following example defines *pws* as a pointer to a structure of type **passwd**, which is used to store the structure pointer returned by the call to the *getpwuid*( ) function. The *geteuid*( ) function shall return the effective user ID of the calling process; this is used as the search criteria for the *getpwuid*( ) function. The call to *getpwuid*( ) shall return a pointer to the structure containing that user ID value.

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
...
struct passwd *pws;
pws = getpwuid(geteuid());
```

Finding an Entry in the User Database

The following example uses *getpwuid*( ) to search the user database for a user ID that was previously stored in a **stat** structure, then prints out the user name if it is found. If the user is not found, the program prints the numeric value of the user ID for the entry.

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
...
struct stat statbuf;
struct passwd *pwd;
...
if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
    printf(" %-8.8s", pwd->pw_name);
else
    printf(" %-8d", statbuf.st_uid);
```

APPLICATION USAGE

Three names associated with the current process can be determined: *getpwuid*(*geteuid*( )) returns the name associated with the effective user ID of the process; *getlogin*( ) returns the name associated with the current login activity; and *getpwuid*(*getuid*( )) returns the name associated with the real user ID of the process.

The *getpwuid_r*( ) function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getpwnam( ), *geteuid*( ), *getuid*( ), *getlogin*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<pwd.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The *getpwuid_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *getpwuid*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *getpwuid_r*( ) function is marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION describing matching the *uid*.

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from *bufsize* characters to bytes.

18100 **NAME**

18101 getrlimit, setrlimit — control maximum resource consumption

18102 **SYNOPSIS**

18103 XSI
```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlp);
int setrlimit(int resource, const struct rlimit *rlp);
```
18104

18105

18106

18107 **DESCRIPTION**

18108 The *getrlimit*( ) function shall get, and the *setrlimit*( ) function shall set, limits on the consumption |
18109 of a variety of resources. |

18110 Each call to either *getrlimit*( ) or *setrlimit*( ) identifies a specific resource to be operated upon as
18111 well as a resource limit. A resource limit is represented by an **rlimit** structure. The *rlim_cur*
18112 member specifies the current or soft limit and the *rlim_max* member specifies the maximum or
18113 hard limit. Soft limits may be changed by a process to any value that is less than or equal to the
18114 hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or
18115 equal to the soft limit. Only a process with appropriate privileges can raise a hard limit. Both
18116 hard and soft limits can be changed in a single call to *setrlimit*( ) subject to the constraints
18117 described above.

18118 The value RLIM_INFINITY, defined in **<sys/resource.h>**, shall be considered to be larger than
18119 any other limit value. If a call to *getrlimit*( ) returns RLIM_INFINITY for a resource, it means the
18120 implementation shall not enforce limits on that resource. Specifying RLIM_INFINITY as any
18121 resource limit value on a successful call to *setrlimit*( ) shall inhibit enforcement of that resource
18122 limit.

18123 The following resources are defined:

18124 RLIMIT_CORE — This is the maximum size of a core file, in bytes, that may be created by a process. A limit of 0 shall prevent the creation of a core file. If this limit is exceeded, the writing of a core file shall terminate at this size. |
18125
18126

18127 RLIMIT_CPU — This is the maximum amount of CPU time, in seconds, used by a process. If this limit is exceeded, SIGXCPU shall be generated for the process. If the process is catching or ignoring SIGXCPU, or all threads belonging to that process are blocking SIGXCPU, the behavior is unspecified. |
18128
18129
18130

18131 RLIMIT_DATA — This is the maximum size of a process' data segment, in bytes. If this limit is exceeded, the *malloc*( ) function shall fail with *errno* set to [ENOMEM]. |
18132

18133 RLIMIT_FSIZE — This is the maximum size of a file, in bytes, that may be created by a process. If a write or truncate operation would cause this limit to be exceeded, SIGXFSZ shall be generated for the thread. If the thread is blocking, or the process is catching or ignoring SIGXFSZ, continued attempts to increase the size of a file from end-of-file to beyond the limit shall fail with *errno* set to [EFBIG]. |
18134
18135
18136
18137
18138

18139 RLIMIT_NOFILE — This is a number one greater than the maximum value that the system may assign to a newly-created descriptor. If this limit is exceeded, functions that allocate new file descriptors may fail with *errno* set to [EMFILE]. This limit constrains the number of file descriptors that a process may allocate.
18140
18141
18142
18143

18144 RLIMIT_STACK — This is the maximum size of a process' stack, in bytes. The implementation does not automatically grow the stack beyond this limit. |
18145

If this limit is exceeded, SIGSEGV shall be generated for the thread. If the thread is blocking SIGSEGV, or the process is ignoring or catching SIGSEGV and has not made arrangements to use an alternate stack, the disposition of SIGSEGV shall be set to SIG_DFL before it is generated.

RLIMIT_AS — This is the maximum size of a process' total available memory, in bytes. If this limit is exceeded, the *malloc*( ) and *mmap*( ) functions shall fail with *errno* set to [ENOMEM]. In addition, the automatic stack growth fails with the effects outlined above.

When using the *getrlimit*( ) function, if a resource limit can be represented correctly in an object of type **rlim_t**, then its representation is returned; otherwise, if the value of the resource limit is equal to that of the corresponding saved hard limit, the value returned shall be RLIM_SAVED_MAX; otherwise, the value returned shall be RLIM_SAVED_CUR.

When using the *setrlimit*( ) function, if the requested new limit is RLIM_INFINITY, the new limit shall be ''no limit''; otherwise, if the requested new limit is RLIM_SAVED_MAX, the new limit shall be the corresponding saved hard limit; otherwise, if the requested new limit is RLIM_SAVED_CUR, the new limit shall be the corresponding saved soft limit; otherwise, the new limit shall be the requested value. In addition, if the corresponding saved limit can be represented correctly in an object of type **rlim_t** then it shall be overwritten with the new limit.

The result of setting a limit to RLIM_SAVED_MAX or RLIM_SAVED_CUR is unspecified unless a previous call to *getrlimit*( ) returned that value as the soft or hard limit for the corresponding resource limit.

The determination of whether a limit can be correctly represented in an object of type **rlim_t** is implementation-defined. For example, some implementations permit a limit whose value is greater than RLIM_INFINITY and others do not.

The *exec* family of functions shall cause resource limits to be saved.

RETURN VALUE

Upon successful completion, *getrlimit*( ) and *setrlimit*( ) shall return 0. Otherwise, these functions shall return –1 and set *errno* to indicate the error.

ERRORS

The *getrlimit*( ) and *setrlimit*( ) functions shall fail if:

[EINVAL] — An invalid *resource* was specified; or in a *setrlimit*( ) call, the new *rlim_cur* exceeds the new *rlim_max*.

[EPERM] — The limit specified to *setrlimit*( ) would have raised the maximum limit value, and the calling process does not have appropriate privileges.

The *setrlimit*( ) function may fail if:

[EINVAL] — The limit specified cannot be lowered because current usage is already higher than the limit.

18183 **EXAMPLES**

18184 None.

18185 **APPLICATION USAGE**

18186 If a process attempts to set the hard limit or soft limit for RLIMIT_NOFILE to less than the value
18187 of {_POSIX_OPEN_MAX} from **<limits.h>**, unexpected behavior may occur.

18188 **RATIONALE**

18189 None.

18190 **FUTURE DIRECTIONS**

18191 None.

18192 **SEE ALSO**

18193 *exec*, *fork*( ), *malloc*( ), *open*( ), *sigaltstack*( ), *sysconf*( ), *ulimit*( ), the Base Definitions volume of
18194 IEEE Std 1003.1-200x, **<stropts.h>**, **<sys/resource.h>**

18195 **CHANGE HISTORY**

18196 First released in Issue 4, Version 2.

18197 **Issue 5**

18198 Moved from X/OPEN UNIX extension to BASE and an APPLICATION USAGE section is added.

18199 Large File Summit extensions are added.

18200 **NAME**

18201 getrusage — get information about resource utilization

18202 **SYNOPSIS**

18203 XSI
```
#include <sys/resource.h>
```

18204
```
int getrusage(int who, struct rusage *r_usage);
```

18205

18206 **DESCRIPTION**

18207 The *getrusage*( ) function shall provide measures of the resources used by the current process or |
18208 its terminated and waited-for child processes. If the value of the *who* argument is |
18209 RUSAGE_SELF, information shall be returned about resources used by the current process. If the
18210 value of the *who* argument is RUSAGE_CHILDREN, information shall be returned about
18211 resources used by the terminated and waited-for children of the current process. If the child is
18212 never waited for (for example, if the parent has SA_NOCLDWAIT set or sets SIGCHLD to
18213 SIG_IGN), the resource information for the child process is discarded and not included in the
18214 resource information provided by *getrusage*( ).

18215 The *r_usage* argument is a pointer to an object of type **struct rusage** in which the returned
18216 information is stored.

18217 **RETURN VALUE**

18218 Upon successful completion, *getrusage*( ) shall return 0; otherwise, –1 shall be returned and *errno*
18219 set to indicate the error.

18220 **ERRORS**

18221 The *getrusage*( ) function shall fail if:

18222 [EINVAL] The value of the *who* argument is not valid.

18223 **EXAMPLES**

18224 **Using getrusage()**

18225 The following example returns information about the resources used by the current process.

18226
```
#include <sys/resource.h>
```
18227
```
...
```
18228
```
int who = RUSAGE_SELF;
```
18229
```
struct rusage usage;
```
18230
```
int ret;
```

18231
```
ret = getrusage(who, &usage);
```

18232 **APPLICATION USAGE**

18233 None.

18234 **RATIONALE**

18235 None.

18236 **FUTURE DIRECTIONS**

18237 None.

18238 **SEE ALSO**

18239 *exit*( ), *sigaction*( ), *time*( ), *times*( ), *wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
18240 **<sys/resource.h>**

18241 **CHANGE HISTORY**

18242 First released in Issue 4, Version 2.

18243 **Issue 5**

18244 Moved from X/OPEN UNIX extension to BASE.

18245 **NAME**

18246 gets — get a string from a stdin stream

18247 **SYNOPSIS**

18248
```
#include <stdio.h>
```

18249
```
char *gets(char *s);
```

18250 **DESCRIPTION**

18251 CX The functionality described on this reference page is aligned with the ISO C standard. Any
18252 conflict between the requirements described here and the ISO C standard is unintentional. This
18253 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

18254 The *gets*( ) function shall read bytes from the standard input stream, *stdin*, into the array pointed
18255 to by *s*, until a newline is read or an end-of-file condition is encountered. Any <newline> shall |
18256 be discarded and a null byte shall be placed immediately after the last byte read into the array. |

18257 CX The *gets*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The
18258 *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ),
18259 *fread*( ), *getc*( ), *getchar*( ), *gets*( ), *fscanf*( ), or *scanf*( ) using *stream* that returns data not supplied by
18260 a prior call to *ungetc*( ).

18261 **RETURN VALUE**

18262 Upon successful completion, *gets*( ) shall return *s*. If the stream is at end-of-file, the end-of-file
18263 indicator for the stream shall be set and *gets*( ) shall return a null pointer. If a read error occurs,
18264 CX the error indicator for the stream shall be set, *gets*( ) shall return a null pointer and set *errno* to
18265 indicate the error.

18266 **ERRORS**

18267 Refer to *fgetc*( ).

18268 **EXAMPLES**

18269 None.

18270 **APPLICATION USAGE**

18271 Reading a line that overflows the array pointed to by *s* results in undefined behavior. The use of
18272 *fgets*( ) is recommended.

18273 Since the user cannot specify the length of the buffer passed to *gets*( ), use of this function is
18274 discouraged. The length of the string read is unlimited. It is possible to overflow this buffer in
18275 such a way as to cause applications to fail, or possible system security violations.

18276 It is recommended that the *fgets*( ) function should be used to read input lines.

18277 **RATIONALE**

18278 None.

18279 **FUTURE DIRECTIONS**

18280 None.

18281 **SEE ALSO**

18282 *feof*( ), *ferror*( ), *fgets*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

18283 **CHANGE HISTORY**

18284 First released in Issue 1. Derived from Issue 1 of the SVID.

18285 **Issue 6**

18286 Extensions beyond the ISO C standard are now marked.

18287 **NAME**

18288 getservbyname — network services database functions

18289 **SYNOPSIS**

18290 `#include <netdb.h>`

18291 `struct servent *getservbyname(const char *`*`name`*`, const char *`*`proto`*`);`

18292 **DESCRIPTION**

18293 Refer to *endservent*( ).

18294 **NAME**

18295 getservbyport — network services database functions

18296 **SYNOPSIS**

18297 `#include <netdb.h>`

18298 `struct servent *getservbyport(int port, const char *proto);`

18299 **DESCRIPTION**

18300 Refer to *endservent*( ).

18301 **NAME**

18302 getservent — network services database functions

18303 **SYNOPSIS**

18304
```
#include <netdb.h>
```

18305
```
struct servent *getservent(void);
```

18306 **DESCRIPTION**

18307 Refer to *endservent*( ).

18308 **NAME**

18309 getsid — get the process group ID of a session leader

18310 **SYNOPSIS**

18311 XSI
```
#include <unistd.h>

pid_t getsid(pid_t pid);
```
18312

18313

18314 **DESCRIPTION**

18315 The *getsid*( ) function shall obtain the process group ID of the process that is the session leader of
18316 the process specified by *pid*. If *pid* is (**pid_t**)0, it specifies the calling process.

18317 **RETURN VALUE**

18318 Upon successful completion, *getsid*( ) shall return the process group ID of the session leader of
18319 the specified process. Otherwise, it shall return (**pid_t**)–1 and set *errno* to indicate the error.

18320 **ERRORS**

18321 The *getsid*( ) function shall fail if:

18322 [EPERM] The process specified by *pid* is not in the same session as the calling process,
18323 and the implementation does not allow access to the process group ID of the
18324 session leader of that process from the calling process.

18325 [ESRCH] There is no process with a process ID equal to *pid*.

18326 **EXAMPLES**

18327 None.

18328 **APPLICATION USAGE**

18329 None.

18330 **RATIONALE**

18331 None.

18332 **FUTURE DIRECTIONS**

18333 None.

18334 **SEE ALSO**

18335 *exec*, *fork*( ), *getpid*( ), *getpgid*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of
18336 IEEE Std 1003.1-200x, **<unistd.h>**

18337 **CHANGE HISTORY**

18338 First released in Issue 4, Version 2.

18339 **Issue 5**

18340 Moved from X/OPEN UNIX extension to BASE.

18341 **NAME**

18342 getsockname — get the socket name

18343 **SYNOPSIS**

18344 `#include <sys/socket.h>`

18345 `int getsockname(int socket, struct sockaddr *restrict address,`
18346 `socklen_t *restrict address_len);`

18347 **DESCRIPTION**

18348 The *getsockname*( ) function shall retrieve the locally-bound name of the specified socket, store
18349 this address in the **sockaddr** structure pointed to by the *address* argument, and store the length of
18350 this address in the object pointed to by the *address_len* argument.

18351 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
18352 the stored address shall be truncated.

18353 If the socket has not been bound to a local name, the value stored in the object pointed to by
18354 *address* is unspecified.

18355 **RETURN VALUE**

18356 Upon successful completion, 0 shall be returned, the *address* argument shall point to the address
18357 of the socket, and the *address_len* argument shall point to the length of the address. Otherwise, –1
18358 shall be returned and *errno* set to indicate the error.

18359 **ERRORS**

18360 The *getsockname*( ) function shall fail if:

18361 [EBADF] The *socket* argument is not a valid file descriptor.

18362 [ENOTSOCK] The *socket* argument does not refer to a socket.

18363 [EOPNOTSUPP] The operation is not supported for this socket's protocol.

18364 The *getsockname*( ) function may fail if:

18365 [EINVAL] The socket has been shut down.

18366 [ENOBUFS] Insufficient resources were available in the system to complete the function.

18367 **EXAMPLES**

18368 None.

18369 **APPLICATION USAGE**

18370 None.

18371 **RATIONALE**

18372 None.

18373 **FUTURE DIRECTIONS**

18374 None.

18375 **SEE ALSO**

18376 *accept*( ), *bind*( ), *getpeername*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
18377 **<sys/socket.h>**

18378 **CHANGE HISTORY**

18379 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

18380 The **restrict** keyword is added to the *getsockname*( ) prototype for alignment with the
18381 ISO / IEC 9899: 1999 standard.

18382 **NAME**

18383 getsockopt — get the socket options

18384 **SYNOPSIS**

18385 `#include <sys/socket.h>`

18386 `int getsockopt(int socket, int level, int option_name,`
18387 `    void *restrict option_value, socklen_t *restrict option_len);`

18388 **DESCRIPTION**

18389 The *getsockopt*( ) function manipulates options associated with a socket.

18390 The *getsockopt*( ) function shall retrieve the value for the option specified by the *option_name*
18391 argument for the socket specified by the *socket* argument. If the size of the option value is greater
18392 than *option_len*, the value stored in the object pointed to by the *option_value* argument shall be
18393 silently truncated. Otherwise, the object pointed to by the *option_len* argument shall be modified
18394 to indicate the actual length of the value.

18395 The *level* argument specifies the protocol level at which the option resides. To retrieve options at
18396 the socket level, specify the *level* argument as SOL_SOCKET. To retrieve options at other levels,
18397 supply the appropriate level identifier for the protocol controlling the option. For example, to
18398 indicate that an option is interpreted by the TCP (Transmission Control Protocol), set *level* to
18399 IPPROTO_TCP as defined in the **<netinet/in.h>** header.

18400 The socket in use may require the process to have appropriate privileges to use the *getsockopt*( )
18401 function.

18402 The *option_name* argument specifies a single option to be retrieved. It can be one of the following
18403 values defined in **<sys/socket.h>**:

18404 SO_DEBUG — Reports whether debugging information is being recorded. This option shall store an **int** value. This is a Boolean option.

18406 SO_ACCEPTCONN — Reports whether socket listening is enabled. This option shall store an **int** value. This is a Boolean option.

18408 SO_BROADCAST — Reports whether transmission of broadcast messages is supported, if this is supported by the protocol. This option shall store an **int** value. This is a Boolean option.

18411 SO_REUSEADDR — Reports whether the rules used in validating addresses supplied to *bind*( ) should allow reuse of local addresses, if this is supported by the protocol. This option shall store an **int** value. This is a Boolean option.

18414 SO_KEEPALIVE — Reports whether connections are kept active with periodic transmission of messages, if this is supported by the protocol.

18416 If the connected socket fails to respond to these messages, the connection shall be broken and threads writing to that socket shall be notified with a SIGPIPE signal. This option shall store an **int** value. This is a Boolean option.

18420 SO_LINGER — Reports whether the socket lingers on *close*( ) if data is present. If SO_LINGER is set, the system blocks the process during *close*( ) until it can transmit the data or until the end of the interval indicated by the *l_linger* member, whichever comes first. If SO_LINGER is not specified, and *close*( ) is issued, the system handles the call in a way that allows the process to continue as quickly as possible. This option shall store a **linger** structure.

SO_OOBINLINE	Reports whether the socket leaves received out-of-band data (data marked urgent) inline. This option shalls tore an **int** value. This is a Boolean option.
SO_SNDBUF	Reports send buffer size information. This option shall store an **int** value.
SO_RCVBUF	Reports receive buffer size information. This option shall store an **int** value.
SO_ERROR	Reports information about error status and clears it. This option shall store an **int** value.
SO_TYPE	Reports the socket type. This option shall store an **int** value. Socket types are described in Section 2.10.6 (on page 509).
SO_DONTROUTE	Reports whether outgoing messages bypass the standard routing facilities. The destination shall be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option shall store an **int** value. This is a Boolean option.
SO_RCVLOWAT	Reports the minimum number of bytes to process for socket input operations. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. (They may return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that returned; for example, out-of-band data.) This option shall store an **int** value. Note that not all implementations allow this option to be retrieved.
SO_RCVTIMEO	Reports the timeout value for input operations. This option shall store a **timeval** structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it shall return with a partial count or *errno* set to [EAGAIN] or [EWOULDBLOCK] if no data was received. The default for this option is zero, which indicates that a receive operation shall not time out. Note that not all implementations allow this option to be retrieved.
SO_SNDLOWAT	Reports the minimum number of bytes to process for socket output operations. Non-blocking output operations shall process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. This option shall store an **int** value. Note that not all implementations allow this option to be retrieved.
SO_SNDTIMEO	Reports the timeout value specifying the amount of time that an output function blocks because flow control prevents data from being sent. If a send operation has blocked for this time, it shall return with a partial count or with *errno* set to [EAGAIN] or [EWOULDBLOCK] if no data were sent. The default for this option is zero, which indicates that a send operation shall not time out. The option shall store a **timeval** structure. Note that not all implementations allow this option to be retrieved.

For Boolean options, a zero value indicates that the option is disabled and a non-zero value indicates that the option is enabled.

Options at other protocol levels vary in format and name.

The socket in use may require the process to have appropriate privileges to use the *getsockopt*( ) function.

RETURN VALUE

Upon successful completion, *getsockopt*( ) shall return 0; otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *getsockopt*( ) function shall fail if:

[EBADF] The *socket* argument is not a valid file descriptor.

[EINVAL] The specified option is invalid at the specified socket level.

[ENOPROTOOPT]
The option is not supported by the protocol.

[ENOTSOCK] The *socket* argument does not refer to a socket.

The *getsockopt*( ) function may fail if:

[EACCES] The calling process does not have the appropriate privileges.

[EINVAL] The socket has been shut down.

[ENOBUFS] Insufficient resources are available in the system to complete the function.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

bind( ), *close*( ), *endprotoent*( ), *setsockopt*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**, **<netinet/in.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the *getsockopt*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

18506 **NAME**

18507 getsubopt — parse suboption arguments from a string

18508 **SYNOPSIS**

18509 XSI
```
#include <stdlib.h>

int getsubopt(char **optionp, char * const *tokens, char **valuep);
```
18510

18511

18512 **DESCRIPTION**

18513 The *getsubopt*( ) function shall parse suboption arguments in a flag argument. Such options often
18514 result from the use of *getopt*( ).

18515 The *getsubopt*( ) argument *optionp* is a pointer to a pointer to the option argument string. The
18516 suboption arguments shall be separated by commas and each may consist of either a single
18517 token, or a token-value pair separated by an equal sign.

18518 The *keylistp* argument shall be a pointer to a vector of strings. The end of the vector is identified
18519 by a null pointer. Each entry in the vector is one of the possible tokens that might be found in
18520 **optionp*. Since commas delimit suboption arguments in *optionp*, they should not appear in any of
18521 the strings pointed to by *keylistp*. Similarly, because an equal sign separates a token from its
18522 value, the application should not include an equal sign in any of the strings pointed to by
18523 *keylistp*.

18524 The *valuep* argument is the address of a value string pointer.

18525 If a comma appears in *optionp*, it shall be interpreted as a suboption separator. After commas
18526 have been processed, if there are one or more equal signs in a suboption string, the first equal
18527 sign in any suboption string shall be interpreted as a separator between a token and a value.
18528 Subsequent equal signs in a suboption string shall be interpreted as part of the value.

18529 If the string at **optionp* contains only one suboption argument (equivalently, no commas),
18530 *getsubopt*( ) shall update **optionp* to point to the nul character at the end of the string. Otherwise,
18531 it shall isolate the suboption argument by replacing the comma separator with a nul character,
18532 and shall update **optionp* to point to the start of the next suboption argument. If the suboption
18533 argument has an associated value (equivalently, contains an equal sign), *getsubopt*( ) shall update
18534 **valuep* to point to the value's first character. Otherwise, it shall set **valuep* to a null pointer. The
18535 calling application may use this information to determine whether the presence or absence of a
18536 value for the suboption is an error.

18537 Additionally, when *getsubopt*( ) fails to match the suboption argument with a token in the *keylistp*
18538 array, the calling application should decide if this is an error, or if the unrecognized option
18539 should be processed in another way.

18540 **RETURN VALUE**

18541 The *getsubopt*( ) function shall return the index of the matched token string, or –1 if no token
18542 strings were matched.

18543 **ERRORS**

18544 No errors are defined.

18545 **EXAMPLES**

```
18546       #include <stdio.h>
18547       #include <stdlib.h>

18548       int do_all;
18549       const char *type;
18550       int read_size;
18551       int write_size;
18552       int read_only;

18553       enum
18554       {
18555           RO_OPTION = 0,
18556           RW_OPTION,
18557           READ_SIZE_OPTION,
18558           WRITE_SIZE_OPTION
18559       };

18560       const char *mount_opts[] =                                          |
18561       {
18562           [RO_OPTION] = "ro",
18563           [RW_OPTION] = "rw",
18564           [READ_SIZE_OPTION] = "rsize",
18565           [WRITE_SIZE_OPTION] = "wsize",
18566           NULL
18567       };

18568       int
18569       main(int argc, char *argv[])
18570       {
18571           char *subopts, *value;
18572           int opt;

18573           while ((opt = getopt(argc, argv, "at:o:")) != -1)
18574               switch(opt)
18575                   {
18576                   case 'a':
18577                       do_all = 1;
18578                       break;
18579                   case 't':
18580                       type = optarg;
18581                       break;
18582                   case 'o':
18583                       subopts = optarg;
18584                       while (*subopts != '\0')                            |
18585                           switch(getsubopt(&subopts, mount_opts, &value)) |
18586                       {
18587                       case RO_OPTION:
18588                           read_only = 1;
18589                           break;
18590                       case RW_OPTION:
18591                           read_only = 0;
18592                           break;
18593                       case READ_SIZE_OPTION:
```

```
                        if (value == NULL)
                            abort();
                        read_size = atoi(value);
                        break;
                    case WRITE_SIZE_OPTION:
                        if (value == NULL)
                            abort();
                        write_size = atoi(value);
                        break;
                    default:
                        /* Unknown suboption. */
                        printf("Unknown suboption ‘%s’\n", value);
                        break;
                    }
                break;
            default:
                abort();
            }

        /* Do the real work. */

        return 0;
    }
```

Parsing Suboptions

The following example uses the *getsubopt*( ) function to parse a value argument in the *optarg* external variable returned by a call to *getopt*( ).

```
    #include <stdlib.h>
    ...
    char *tokens[] = {"HOME", "PATH", "LOGNAME", (char *) NULL };
    char *value;
    int opt, index;

    while ((opt = getopt(argc, argv, "e:")) != -1) {
        switch(opt)  {
        case ’e’ :
            while ((index = getsubopt(&optarg, tokens, &value)) != -1) {
                switch(index) {
    ...
                }
                break;
    ...
            }
    }
    ...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getopt( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

18647 **NAME**

18648 gettimeofday — get the date and time

18649 **SYNOPSIS**

18650 XSI
```
#include <sys/time.h>

int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
```
18651

18652

18653 **DESCRIPTION**

18654 The *gettimeofday*( ) function shall obtain the current time, expressed as seconds and |
18655 microseconds since the Epoch, and store it in the **timeval** structure pointed to by *tp*. The |
18656 resolution of the system clock is unspecified.

18657 If *tzp* is not a null pointer, the behavior is unspecified.

18658 **RETURN VALUE**

18659 The *gettimeofday*( ) function shall return 0 and no value shall be reserved to indicate an error.

18660 **ERRORS**

18661 No errors are defined.

18662 **EXAMPLES**

18663 None.

18664 **APPLICATION USAGE**

18665 None.

18666 **RATIONALE**

18667 None.

18668 **FUTURE DIRECTIONS**

18669 None.

18670 **SEE ALSO**

18671 *ctime*( ), *ftime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/time.h>**

18672 **CHANGE HISTORY**

18673 First released in Issue 4, Version 2.

18674 **Issue 5**

18675 Moved from X/OPEN UNIX extension to BASE.

18676 **Issue 6**

18677 The DESCRIPTION is updated to refer to ‘‘seconds since the Epoch’’ rather than ‘‘seconds since
18678 00:00:00 UTC (Coordinated Universal Time), January 1 1970’’ for consistency with other *time*
18679 functions.

18680 The **restrict** keyword is added to the *gettimeofday*( ) prototype for alignment with the
18681 ISO/IEC 9899: 1999 standard.

18682 **NAME**

18683 getuid — get a real user ID

18684 **SYNOPSIS**

18685
```
#include <unistd.h>
```

18686
```
uid_t getuid(void);
```

18687 **DESCRIPTION**

18688 The *getuid*( ) function shall return the real user ID of the calling process.

18689 **RETURN VALUE**

18690 The *getuid*( ) function shall always be successful and no return value is reserved to indicate the |
18691 error.

18692 **ERRORS**

18693 No errors are defined.

18694 **EXAMPLES**

18695 **Setting the Effective User ID to the Real User ID**

18696 The following example sets the effective user ID and the real user ID of the current process to the
18697 real user ID of the caller.

18698
```
#include <unistd.h>
#include <sys/types.h>
...
setreuid(getuid(), getuid());
...
```

18703 **APPLICATION USAGE**

18704 None.

18705 **RATIONALE**

18706 None.

18707 **FUTURE DIRECTIONS**

18708 None.

18709 **SEE ALSO**

18710 *getegid*( ), *geteuid*( ), *getgid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base
18711 Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

18712 **CHANGE HISTORY**

18713 First released in Issue 1. Derived from Issue 1 of the SVID.

18714 **Issue 6**

18715 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

18716 The following new requirements on POSIX implementations derive from alignment with the
18717 Single UNIX Specification:

18718 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
18719 required for conforming implementations of previous POSIX specifications, it was not
18720 required for UNIX applications.

18721 **NAME**

18722 getutxent, getutxid, getutxline — get user accounting database entries

18723 **SYNOPSIS**

18724 XSI
```
#include <utmpx.h>

struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *id);
struct utmpx *getutxline(const struct utmpx *line);
```
18725

18726

18727

18728

18729 **DESCRIPTION**

18730 Refer to *endutxent*( ).

18731 **NAME**

18732 getwc — get a wide character from a stream

18733 **SYNOPSIS**

18734 `#include <stdio.h>`
18735 `#include <wchar.h>`

18736 `wint_t getwc(FILE *stream);`

18737 **DESCRIPTION**

18738 CX The functionality described on this reference page is aligned with the ISO C standard. Any
18739 conflict between the requirements described here and the ISO C standard is unintentional. This
18740 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

18741 The *getwc*( ) function shall be equivalent to *fgetwc*( ), except that if it is implemented as a macro it |
18742 may evaluate *stream* more than once, so the argument should never be an expression with side
18743 effects.

18744 **RETURN VALUE**

18745 Refer to *fgetwc*( ).

18746 **ERRORS**

18747 Refer to *fgetwc*( ).

18748 **EXAMPLES**

18749 None.

18750 **APPLICATION USAGE**

18751 Since it may be implemented as a macro, *getwc*( ) may treat incorrectly a *stream* argument with |
18752 side effects. In particular, *getwc*(*f*++) does not necessarily work as expected. Therefore, use of
18753 this function is not recommended; *fgetwc*( ) should be used instead.

18754 **RATIONALE**

18755 None.

18756 **FUTURE DIRECTIONS**

18757 None.

18758 **SEE ALSO**

18759 *fgetwc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

18760 **CHANGE HISTORY**

18761 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
18762 draft.

18763 **Issue 5**

18764 The Optional Header (OH) marking is removed from **<stdio.h>**.

18765 **NAME**

18766 getwchar — get a wide character from a stdin stream

18767 **SYNOPSIS**

18768
```
#include <wchar.h>
```

18769
```
wint_t getwchar(void);
```

18770 **DESCRIPTION**

18771 CX The functionality described on this reference page is aligned with the ISO C standard. Any
18772 conflict between the requirements described here and the ISO C standard is unintentional. This
18773 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

18774 The *getwchar*( ) function shall be equivalent to *getwc*(*stdin*).

18775 **RETURN VALUE**

18776 Refer to *fgetwc*( ).

18777 **ERRORS**

18778 Refer to *fgetwc*( ).

18779 **EXAMPLES**

18780 None.

18781 **APPLICATION USAGE**

18782 If the **wint_t** value returned by *getwchar*( ) is stored into a variable of type **wchar_t** and then
18783 compared against the **wint_t** macro WEOF, the result may be incorrect. Only the **wint_t** type is
18784 guaranteed to be able to represent any wide character and WEOF.

18785 **RATIONALE**

18786 None.

18787 **FUTURE DIRECTIONS**

18788 None.

18789 **SEE ALSO**

18790 *fgetwc*( ), *getwc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

18791 **CHANGE HISTORY**

18792 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
18793 draft.

18794 **NAME**

18795 getwd — get the current working directory pathname (**LEGACY**)

18796 **SYNOPSIS**

18797 XSI
```
#include <unistd.h>

char *getwd(char *path_name);
```
18798

18799

18800 **DESCRIPTION**

18801 The *getwd*( ) function shall determine an absolute pathname of the current working directory of
18802 the calling process, and copy a string containing that pathname into the array pointed to by the
18803 *path_name* argument.

18804 If the length of the pathname of the current working directory is greater than ({PATH_MAX}+1)
18805 including the null byte, *getwd*( ) shall fail and return a null pointer.

18806 **RETURN VALUE**

18807 Upon successful completion, a pointer to the string containing the absolute pathname of the
18808 current working directory shall be returned. Otherwise, *getwd*( ) shall return a null pointer and
18809 the contents of the array pointed to by *path_name* are undefined.

18810 **ERRORS**

18811 No errors are defined.

18812 **EXAMPLES**

18813 None.

18814 **APPLICATION USAGE**

18815 For applications portability, the *getcwd*( ) function should be used to determine the current
18816 working directory instead of *getwd*( ).

18817 **RATIONALE**

18818 Since the user cannot specify the length of the buffer passed to *getwd*( ), use of this function is
18819 discouraged. The length of a pathname described in {PATH_MAX} is file system-dependent and
18820 may vary from one mount point to another, or might even be unlimited. It is possible to
18821 overflow this buffer in such a way as to cause applications to fail, or possible system security
18822 violations.

18823 It is recommended that the *getcwd*( ) function should be used to determine the current working
18824 directory.

18825 **FUTURE DIRECTIONS**

18826 This function may be withdrawn in a future version.

18827 **SEE ALSO**

18828 *getcwd*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

18829 **CHANGE HISTORY**

18830 First released in Issue 4, Version 2.

18831 **Issue 5**

18832 Moved from X/OPEN UNIX extension to BASE.

18833 **Issue 6**

18834 This function is marked LEGACY.

18835 **NAME**

18836 glob, globfree — generate pathnames matching a pattern

18837 **SYNOPSIS**

18838
```
#include <glob.h>
```

18839
```
int glob(const char *restrict pattern, int flags,
    int(*errfunc)(const char *epath, int eerrno),
    glob_t *restrict pglob);
void globfree(glob_t *pglob);
```

18843 **DESCRIPTION**

18844 The *glob*( ) function is a pathname generator that shall implement the rules defined in the Shell
18845 and Utilities volume of IEEE Std 1003.1-200x, Section 2.13, Pattern Matching Notation, with
18846 optional support for rule 3 in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section
18847 2.13.3, Patterns Used for Filename Expansion.

18848 The structure type **glob_t** is defined in **<glob.h>** and includes at least the following members:

18849

18850

Member Type	Member Name	Description
size_t	*gl_pathc*	Count of paths matched by *pattern*.
char **	*gl_pathv*	Pointer to a list of matched pathnames.
size_t	*gl_offs*	Slots to reserve at the beginning of *gl_pathv*.

18854 The argument *pattern* is a pointer to a pathname pattern to be expanded. The *glob*( ) function
18855 shall match all accessible pathnames against this pattern and develop a list of all pathnames that
18856 match. In order to have access to a pathname, *glob*( ) requires search permission on every
18857 component of a path except the last, and read permission on each directory of any filename
18858 component of *pattern* that contains any of the following special characters: `'*'`, `'?'`, and `'['`.

18859 The *glob*( ) function shall store the number of matched pathnames into *pglob->gl_pathc* and a
18860 pointer to a list of pointers to path names into *pglob->gl_pathv*. The pathnames shall be in sort
18861 order as defined by the current setting of the *LC_COLLATE* category; see the Base Definitions
18862 volume of IEEE Std 1003.1-200x, Section 7.3.2, LC_COLLATE. The first pointer after the last
18863 pathname shall be a null pointer. If the pattern does not match any pathnames, the returned
18864 number of matched paths is set to 0, and the contents of *pglob->gl_pathv* are implementation-
18865 defined.

18866 It is the caller's responsibility to create the structure pointed to by *pglob*. The *glob*( ) function shall
18867 allocate other space as needed, including the memory pointed to by *gl_pathv*. The *globfree*( )
18868 function shall free any space associated with *pglob* from a previous call to *glob*( ).

18869 The *flags* argument is used to control the behavior of *glob*( ). The value of *flags* is a bitwise-
18870 inclusive OR of zero or more of the following constants, which are defined in **<glob.h>**:

18871 GLOB_APPEND — Append pathnames generated to the ones from a previous call to *glob*( ).

18872 GLOB_DOOFFS — Make use of *pglob->gl_offs*. If this flag is set, *pglob->gl_offs* is used to
18873 specify how many null pointers to add to the beginning of *pglob->gl_pathv*. In other words, *pglob->gl_pathv* shall point to *pglob->gl_offs* null
18875 pointers, followed by *pglob->gl_pathc* pathname pointers, followed by a
18876 null pointer.

18877 GLOB_ERR — Cause *glob*( ) to return when it encounters a directory that it cannot open
18878 or read. Ordinarily, *glob*( ) continues to find matches.

GLOB_MARK	Each pathname that is a directory that matches *pattern* shall have a slash appended.	
GLOB_NOCHECK	Supports rule 3 in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13.3, Patterns Used for Filename Expansion. If *pattern* does not match any pathname, then *glob*( ) shall return a list consisting of only *pattern*, and the number of matched pathnames is 1.	
GLOB_NOESCAPE	Disable backslash escaping.	
GLOB_NOSORT	Ordinarily, *glob*( ) sorts the matching pathnames according to the current setting of the *LC_COLLATE* category, see the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.2, LC_COLLATE . When this flag is used, the order of path names returned is unspecified.	

The GLOB_APPEND flag can be used to append a new set of pathnames to those found in a previous call to *glob*( ). The following rules apply to applications when two or more calls to *glob*( ) are made with the same value of *pglob* and without intervening calls to *globfree*( ):

1. The first such call shall not set GLOB_APPEND. All subsequent calls shall set it.
2. All the calls shall set GLOB_DOOFFS, or all shall not set it.
3. After the second call, *pglob->gl_pathv* points to a list containing the following:
   a. Zero or more null pointers, as specified by GLOB_DOOFFS and *pglob->gl_offs*.
   b. Pointers to the pathnames that were in the *pglob->gl_pathv* list before the call, in the same order as before.
   c. Pointers to the new pathnames generated by the second call, in the specified order.
4. The count returned in *pglob->gl_pathc* shall be the total number of pathnames from the two calls.
5. The application can change any of the fields after a call to *glob*( ). If it does, the application shall reset them to the original value before a subsequent call, using the same *pglob* value, to *globfree*( ) or *glob*( ) with the GLOB_APPEND flag.

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is not a null pointer, *glob*( ) calls (**errfunc*( )) with two arguments:

1. The *epath* argument is a pointer to the path that failed.
2. The *eerrno* argument is the value of *errno* from the failure, as set by *opendir*( ), *readdir*( ), or *stat*( ). (Other values may be used to report other errors not explicitly documented for those functions.)

If (**errfunc*( )) is called and returns non-zero, or if the GLOB_ERR flag is set in *flags*, *glob*( ) shall stop the scan and return GLOB_ABORTED after setting *gl_pathc* and *gl_pathv* in *pglob* to reflect the paths already scanned. If GLOB_ERR is not set and either *errfunc* is a null pointer or (**errfunc*( )) returns 0, the error shall be ignored.

The *glob*( ) function shall not fail because of large files.

RETURN VALUE

Upon successful completion, *glob*( ) shall return 0. The argument *pglob->gl_pathc* shall return the number of matched pathnames and the argument *pglob->gl_pathv* shall contain a pointer to a null-terminated list of matched and sorted pathnames. However, if *pglob->gl_pathc* is 0, the content of *pglob->gl_pathv* is undefined.

The *globfree*( ) function shall not return a value.

If *glob*( ) terminates due to an error, it shall return one of the non-zero constants defined in **<glob.h>**. The arguments *pglob->gl_pathc* and *pglob->gl_pathv* are still set as defined above.

ERRORS

The *glob*( ) function shall fail and return the corresponding value if:

GLOB_ABORTED — The scan was stopped because GLOB_ERR was set or (**errfunc*( )) returned non-zero.

GLOB_NOMATCH — The pattern does not match any existing pathname, and GLOB_NOCHECK was not set in flags.

GLOB_NOSPACE — An attempt to allocate memory failed.

EXAMPLES

One use of the GLOB_DOOFFS flag is by applications that build an argument list for use with *execv*( ), *execve*( ), or *execvp*( ). Suppose, for example, that an application wants to do the equivalent of:

```
ls -l *.c
```

but for some reason:

```
system("ls -l *.c")
```

is not acceptable. The application could obtain approximately the same result using the sequence:

```
globbuf.gl_offs = 2;
glob("*.c", GLOB_DOOFFS, NULL, &globbuf);
globbuf.gl_pathv[0] = "ls";
globbuf.gl_pathv[1] = "-l";
execvp("ls", &globbuf.gl_pathv[0]);
```

Using the same example:

```
ls -l *.c *.h
```

could be approximately simulated using GLOB_APPEND as follows:

```
globbuf.gl_offs = 2;
glob("*.c", GLOB_DOOFFS, NULL, &globbuf);
glob("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);
...
```

APPLICATION USAGE

This function is not provided for the purpose of enabling utilities to perform pathname expansion on their arguments, as this operation is performed by the shell, and utilities are explicitly not expected to redo this. Instead, it is provided for applications that need to do pathname expansion on strings obtained from other sources, such as a pattern typed by a user or read from a file.

If a utility needs to see if a pathname matches a given pattern, it can use *fnmatch*( ).

Note that *gl_pathc* and *gl_pathv* have meaning even if *glob*( ) fails. This allows *glob*( ) to report partial results in the event of an error. However, if *gl_pathc* is 0, *gl_pathv* is unspecified even if *glob*( ) did not return an error.

The GLOB_NOCHECK option could be used when an application wants to expand a pathname if wildcards are specified, but wants to treat the pattern as just a string otherwise. The *sh* utility

might use this for option-arguments, for example.

The new pathnames generated by a subsequent call with GLOB_APPEND are not sorted together with the previous pathnames. This mirrors the way that the shell handles pathname expansion when multiple expansions are done on a command line.

Applications that need tilde and parameter expansion should use *wordexp*( ).

RATIONALE

It was claimed that the GLOB_DOOFFS flag is unnecessary because it could be simulated using:

```
new = (char **)malloc((n + pglob->gl_pathc + 1)
       * sizeof(char *));
(void) memcpy(new+n, pglob->gl_pathv,
       pglob->gl_pathc * sizeof(char *));
(void) memset(new, 0, n * sizeof(char *));
free(pglob->gl_pathv);
pglob->gl_pathv = new;
```

However, this assumes that the memory pointed to by *gl_pathv* is a block that was separately created using *malloc*( ). This is not necessarily the case. An application should make no assumptions about how the memory referenced by fields in *pglob* was allocated. It might have been obtained from *malloc*( ) in a large chunk and then carved up within *glob*( ), or it might have been created using a different memory allocator. It is not the intent of the standard developers to specify or imply how the memory used by *glob*( ) is managed.

The GLOB_APPEND flag would be used when an application wants to expand several different patterns into a single list.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fnmatch*( ), *opendir*( ), *readdir*( ), *stat*( ), *wordexp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<glob.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

Moved from POSIX2 C-language Binding to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *glob*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

18999 **NAME**

19000 gmtime, gmtime_r — convert a time value to a broken-down UTC time

19001 **SYNOPSIS**

19002 `#include <time.h>`

19003 `struct tm *gmtime(const time_t *timer);`

19004 TSF `struct tm *gmtime_r(const time_t *restrict timer,`

19005 `struct tm *restrict result);`

19006

19007 **DESCRIPTION**

19008 CX For *gmtime*( ): The functionality described on this reference page is aligned with the ISO C
19009 standard. Any conflict between the requirements described here and the ISO C standard is
19010 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

19011 The *gmtime*( ) function shall convert the time in seconds since the Epoch pointed to by *timer* into
19012 a broken-down time, expressed as Coordinated Universal Time (UTC).

19013 CX The relationship between a time in seconds since the Epoch used as an argument to *gmtime*( ) |
19014 and the **tm** structure (defined in the **<time.h>** header) is that the result shall be as specified in the |
19015 expression given in the definition of seconds since the Epoch (see the Base Definitions volume of |
19016 IEEE Std 1003.1-200x, Section 4.14, Seconds Since the Epoch), where the names in the structure |
19017 and in the expression correspond. |

19018 TSF The same relationship shall apply for *gmtime_r*( ). |

19019 CX The *gmtime*( ) function need not be reentrant. A function that is not required to be reentrant is not |
19020 required to be thread-safe. |

19021 The *asctime*( ), *ctime*( ), *gmtime*( ), and *localtime*( ) functions shall return values in one of two static |
19022 objects: a broken-down time structure and an array of type **char**. Execution of any of the |
19023 functions may overwrite the information returned in either of these objects by any of the other |
19024 functions. |

19025 TSF The *gmtime_r*( ) function shall convert the time in seconds since the Epoch pointed to by *timer*
19026 into a broken-down time expressed as Coordinated Universal Time (UTC). The broken-down
19027 time is stored in the structure referred to by *result*. The *gmtime_r*( ) function shall also return the
19028 address of the same structure.

19029 **RETURN VALUE**

19030 The *gmtime*( ) function shall return a pointer to a **struct tm**.

19031 TSF Upon successful completion, *gmtime_r*( ) shall return the address of the structure pointed to by
19032 the argument *result*. If an error is detected, or UTC is not available, *gmtime_r*( ) shall return a null
19033 pointer.

19034 **ERRORS**

19035 No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *gmtime_r*( ) function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime( ), *clock*( ), *ctime*( ), *difftime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A note indicating that the *gmtime*( ) function need not be reentrant is added to the DESCRIPTION.

The *gmtime_r*( ) function is included for alignment with the POSIX Threads Extension.

Issue 6

The *gmtime_r*( ) function is marked as part of the Thread-Safe Functions option.

Extensions beyond the ISO C standard are now marked.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The **restrict** keyword is added to the *gmtime_r*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

19061 **NAME**

19062 grantpt — grant access to the slave pseudo-terminal device

19063 **SYNOPSIS**

19064 XSI
```
#include <stdlib.h>

int grantpt(int fildes);
```
19065

19066

19067 **DESCRIPTION**

19068 The *grantpt*( ) function shall change the mode and ownership of the slave pseudo-terminal
19069 device associated with its master pseudo-terminal counterpart. The *fildes* argument is a file
19070 descriptor that refers to a master pseudo-terminal device. The user ID of the slave shall be set to
19071 the real UID of the calling process and the group ID shall be set to an unspecified group ID. The
19072 permission mode of the slave pseudo-terminal shall be set to readable and writable by the
19073 owner, and writable by the group.

19074 The behavior of the *grantpt*( ) function is unspecified if the application has installed a signal
19075 handler to catch SIGCHLD signals.

19076 **RETURN VALUE**

19077 Upon successful completion, *grantpt*( ) shall return 0; otherwise, it shall return –1 and set *errno* to
19078 indicate the error.

19079 **ERRORS**

19080 The *grantpt*( ) function may fail if:

19081 [EBADF] The *fildes* argument is not a valid open file descriptor.

19082 [EINVAL] The *fildes* argument is not associated with a master pseudo-terminal device.

19083 [EACCES] The corresponding slave pseudo-terminal device could not be accessed.

19084 **EXAMPLES**

19085 None.

19086 **APPLICATION USAGE**

19087 None.

19088 **RATIONALE**

19089 None.

19090 **FUTURE DIRECTIONS**

19091 None.

19092 **SEE ALSO**

19093 *open*( ), *ptsname*( ), *unlockpt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

19094 **CHANGE HISTORY**

19095 First released in Issue 4, Version 2.

19096 **Issue 5**

19097 Moved from X/OPEN UNIX extension to BASE.

19098 The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section in
19099 previous issues.

19100 **NAME**

19101 h_errno — error return value for network database operations

19102 **SYNOPSIS**

19103 OB
```
#include <netdb.h>
```
19104

19105 **DESCRIPTION**

19106 Note that this method of returning errors is used only in connection with obsolescent functions.

19107 The **<netdb.h>** header provides a declaration of *h_errno* as a modifiable *l*-value of type **int**.

19108 It is unspecified whether *h_errno* is a macro or an identifier declared with external linkage. If a
19109 macro definition is suppressed in order to access an actual object, or a program defines an
19110 identifier with the name *h_errno*, the behavior is undefined.

19111 **RETURN VALUE**

19112 None.

19113 **ERRORS**

19114 No errors are defined.

19115 **EXAMPLES**

19116 None.

19117 **APPLICATION USAGE**

19118 Applications should obtain the definition of *h_errno* by the inclusion of the **<netdb.h>** header.

19119 **RATIONALE**

19120 None.

19121 **FUTURE DIRECTIONS**

19122 *h_errno* may be withdrawn in a future version.

19123 **SEE ALSO**

19124 *endhostent*( ), *errno*, the Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

19125 **CHANGE HISTORY**

19126 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19127 **NAME**

19128 hcreate, hdestroy, hsearch — manage hash search table

19129 **SYNOPSIS**

19130 XSI
```
#include <search.h>

int hcreate(size_t nel);
void hdestroy(void);
ENTRY *hsearch(ENTRY item, ACTION action);
```
19134

19135 **DESCRIPTION**

19136 The *hcreate*( ), *hdestroy*( ), and *hsearch*( ) functions shall manage hash search tables.

19137 The *hcreate*( ) function shall allocate sufficient space for the table, and the application shall
19138 ensure it is called before *hsearch*( ) is used. The *nel* argument is an estimate of the maximum
19139 number of entries that the table shall contain. This number may be adjusted upward by the
19140 algorithm in order to obtain certain mathematically favorable circumstances.

19141 The *hdestroy*( ) function shall dispose of the search table, and may be followed by another call to
19142 *hcreate*( ). After the call to *hdestroy*( ), the data can no longer be considered accessible.

19143 The *hsearch*( ) function is a hash-table search routine. It shall return a pointer into a hash table
19144 indicating the location at which an entry can be found. The *item* argument is a structure of type
19145 **ENTRY** (defined in the **<search.h>** header) containing two pointers: *item.key* points to the
19146 comparison key (a **char ***), and *item.data* (a **void ***) points to any other data to be associated with
19147 that key. The comparison function used by *hsearch*( ) is *strcmp*( ). The *action* argument is a
19148 member of an enumeration type **ACTION** indicating the disposition of the entry if it cannot be
19149 found in the table. ENTER indicates that the item should be inserted in the table at an
19150 appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is
19151 indicated by the return of a null pointer.

19152 These functions need not be reentrant. A function that is not required to be reentrant is not
19153 required to be thread-safe.

19154 **RETURN VALUE**

19155 The *hcreate*( ) function shall return 0 if it cannot allocate sufficient space for the table; otherwise,
19156 it shall return non-zero.

19157 The *hdestroy*( ) function shall not return a value.

19158 The *hsearch*( ) function shall return a null pointer if either the action is FIND and the item could
19159 not be found or the action is ENTER and the table is full.

19160 **ERRORS**

19161 The *hcreate*( ) and *hsearch*( ) functions may fail if:

19162 [ENOMEM] Insufficient storage space is available.

19163 **EXAMPLES**

19164 The following example reads in strings followed by two numbers and stores them in a hash
19165 table, discarding duplicates. It then reads in strings and finds the matching entry in the hash
19166 table and prints it out.

```
#include <stdio.h>
#include <search.h>
#include <string.h>

struct info {        /* This is the info stored in the table */
    int age, room;   /* other than the key. */
```

```
};
#define NUM_EMPL    5000    /* # of elements in search table. */
int main(void)
{
    char string_space[NUM_EMPL*20];   /* Space to store strings. */
    struct info info_space[NUM_EMPL]; /* Space to store employee info. */
    char *str_ptr = string_space;     /* Next space in string_space. */
    struct info *info_ptr = info_space;
                                      /* Next space in info_space. */
    ENTRY item;
    ENTRY *found_item; /* Name to look for in table. */
    char name_to_find[30];

    int i = 0;

    /* Create table; no error checking is performed. */
    (void) hcreate(NUM_EMPL);
    while (scanf("%s%d%d", str_ptr, &info_ptr->age,
           &info_ptr->room) != EOF && i++ < NUM_EMPL) {

        /* Put information in structure, and structure in item. */
        item.key = str_ptr;
        item.data = info_ptr;
        str_ptr += strlen(str_ptr) + 1;
        info_ptr++;

        /* Put item into table. */
        (void) hsearch(item, ENTER);
    }

    /* Access table. */
    item.key = name_to_find;
    while (scanf("%s", item.key) != EOF) {
        if ((found_item = hsearch(item, FIND)) != NULL) {

            /* If item is in the table. */
            (void)printf("found %s, age = %d, room = %d\n",
                found_item->key,
                ((struct info *)found_item->data)->age,
                ((struct info *)found_item->data)->room);
        } else
            (void)printf("no such employee %s\n", name_to_find);
    }
    return 0;
}
```

APPLICATION USAGE

The *hcreate*( ) and *hsearch*( ) functions may use *malloc*( ) to allocate space.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

bsearch( ), *lsearch*( ), *malloc*( ), *strcmp*( ), *tsearch*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<search.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

19225 **NAME**

19226 hdestroy — manage hash search table

19227 **SYNOPSIS**

19228 XSI
```
#include <search.h>
```

19229
```
void hdestroy(void);
```

19230

19231 **DESCRIPTION**

19232 Refer to *hcreate*( ).

19233 **NAME**

19234 hsearch — manage hash search table

19235 **SYNOPSIS**

19236 XSI
```
#include <search.h>
```

19237
```
ENTRY *hsearch(ENTRY item, ACTION action);
```

19238

19239 **DESCRIPTION**

19240 Refer to *hcreate*( ).

19241 **NAME**

19242 htonl, htons, ntohl, ntohs — convert values between host and network byte order

19243 **SYNOPSIS**

19244 ```
#include <arpa/inet.h>
```

19245 ```
uint32_t htonl(uint32_t hostlong);
```
19246 ```
uint16_t htons(uint16_t hostshort);
```
19247 ```
uint32_t ntohl(uint32_t netlong);
```
19248 ```
uint16_t ntohs(uint16_t netshort);
```

19249 **DESCRIPTION**

19250 These functions shall convert 16-bit and 32-bit quantities between network byte order and host
19251 byte order.

19252 On some implementations, these functions are defined as macros.

19253 The **uint32_t** and **uint16_t** types are defined in **<inttypes.h>**. |

19254 **RETURN VALUE**

19255 The *htonl*( ) and *htons*( ) functions shall return the argument value converted from host to
19256 network byte order.

19257 The *ntohl*( ) and *ntohs*( ) functions shall return the argument value converted from network to
19258 host byte order.

19259 **ERRORS**

19260 No errors are defined.

19261 **EXAMPLES**

19262 None.

19263 **APPLICATION USAGE**

19264 These functions are most often used in conjunction with IPv4 addresses and ports as returned by
19265 *gethostent*( ) and *getservent*( ).

19266 **RATIONALE**

19267 None.

19268 **FUTURE DIRECTIONS**

19269 None.

19270 **SEE ALSO**

19271 *endhostent*( ), *endservent*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<inttypes.h>**,
19272 **<arpa/inet.h>**

19273 **CHANGE HISTORY**

19274 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
```

19275 **NAME**

19276 htons — convert values between host and network byte order

19277 **SYNOPSIS**

19278 `#include <arpa/inet.h>`

19279 `uint16_t htons(uint16_t hostshort);`

19280 **DESCRIPTION**

19281 Refer to *htonl*( ).

19282 **NAME**

19283 hypot, hypotf, hypotl — Euclidean distance function

19284 **SYNOPSIS**

19285 ```
#include <math.h>
```

19286 ```
double hypot(double x, double y);
```
19287 ```
float hypotf(float x, float y);
```
19288 ```
long double hypotl(long double x, long double y);
```

19289 **DESCRIPTION**

19290 CX The functionality described on this reference page is aligned with the ISO C standard. Any
19291 conflict between the requirements described here and the ISO C standard is unintentional. This
19292 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

19293 These functions shall compute the value of the square root of x^2+y^2 without undue overflow or |
19294 underflow. |

19295 An application wishing to check for error situations should set *errno* to zero and call
19296 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
19297 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
19298 zero, an error has occurred.

19299 **RETURN VALUE**

19300 Upon successful completion, these functions shall return the length of the hypotenuse of a
19301 right-angled triangle with sides of length *x* and *y*.

19302 If the correct value would cause overflow, a range error shall occur and *hypot*( ), *hypotf*( ), and
19303 *hypotl*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
19304 respectively.

19305 MX If *x* or *y* is ±Inf, +Inf shall be returned (even if one of *x* or *y* is NaN).

19306 If *x* or *y* is NaN, and the other is not ±Inf, a NaN shall be returned.

19307 If both arguments are subnormal and the correct result is subnormal, a range error may occur
19308 and the correct result is returned.

19309 **ERRORS**

19310 These functions shall fail if:

19311 Range Error The result overflows.

19312 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
19313 then *errno* shall be set to [ERANGE]. If the integer expression |
19314 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow |
19315 floating-point exception shall be raised. |

19316 These functions may fail if:

19317 MX Range Error The result underflows.

19318 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
19319 then *errno* shall be set to [ERANGE]. If the integer expression |
19320 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow |
19321 floating-point exception shall be raised. |

EXAMPLES

None.

APPLICATION USAGE

hypot(*x*,*y*), *hypot*(*y*,*x*), and *hypot*(*x*, –*y*) are equivalent.

hypot(*x*, ±0) is equivalent to *fabs*(*x*).

Underflow only happens when both *x* and *y* are subnormal and the (inexact) result is also subnormal.

These functions take precautions against overflow during intermediate steps of the computation.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *isnan*( ), *sqrt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *hypot*( ) function is no longer marked as an extension.

The *hypotf*( ) and *hypotl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

19353 **NAME**

19354 iconv — codeset conversion function

19355 **SYNOPSIS**

19356 XSI
```
#include <iconv.h>

size_t iconv(iconv_t cd, char **restrict inbuf,
    size_t *restrict inbytesleft, char **restrict outbuf,
    size_t *restrict outbytesleft);
```
19360

19361 **DESCRIPTION**

19362 The *iconv*( ) function shall convert the sequence of characters from one codeset, in the array
19363 specified by *inbuf*, into a sequence of corresponding characters in another codeset, in the array
19364 specified by *outbuf*. The codesets are those specified in the *iconv_open*( ) call that returned the
19365 conversion descriptor, *cd*. The *inbuf* argument points to a variable that points to the first
19366 character in the input buffer and *inbytesleft* indicates the number of bytes to the end of the buffer
19367 to be converted. The *outbuf* argument points to a variable that points to the first available byte in
19368 the output buffer and *outbytesleft* indicates the number of the available bytes to the end of the
19369 buffer.

19370 For state-dependent encodings, the conversion descriptor *cd* is placed into its initial shift state by
19371 a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When *iconv*( ) is
19372 called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft*
19373 points to a positive value, *iconv*( ) shall place, into the output buffer, the byte sequence to change
19374 the output buffer to its initial shift state. If the output buffer is not large enough to hold the
19375 entire reset sequence, *iconv*( ) shall fail and set *errno* to [E2BIG]. Subsequent calls with *inbuf* as
19376 other than a null pointer or a pointer to a null pointer cause the conversion to take place from
19377 the current state of the conversion descriptor.

19378 If a sequence of input bytes does not form a valid character in the specified codeset, conversion |
19379 shall stop after the previous successfully converted character. If the input buffer ends with an |
19380 incomplete character or shift sequence, conversion shall stop after the previous successfully |
19381 converted bytes. If the output buffer is not large enough to hold the entire converted input, |
19382 conversion shall stop just prior to the input bytes that would cause the output buffer to |
19383 overflow. The variable pointed to by *inbuf* shall be updated to point to the byte following the last |
19384 byte successfully used in the conversion. The value pointed to by *inbytesleft* shall be |
19385 decremented to reflect the number of bytes still not converted in the input buffer. The variable |
19386 pointed to by *outbuf* shall be updated to point to the byte following the last byte of converted |
19387 output data. The value pointed to by *outbytesleft* shall be decremented to reflect the number of |
19388 bytes still available in the output buffer. For state-dependent encodings, the conversion |
19389 descriptor shall be updated to reflect the shift state in effect at the end of the last successfully |
19390 converted byte sequence.

19391 If *iconv*( ) encounters a character in the input buffer that is valid, but for which an identical
19392 character does not exist in the target codeset, *iconv*( ) shall perform an implementation-defined |
19393 conversion on this character. |

19394 **RETURN VALUE**

19395 The *iconv*( ) function shall update the variables pointed to by the arguments to reflect the extent
19396 of the conversion and return the number of non-identical conversions performed. If the entire
19397 string in the input buffer is converted, the value pointed to by *inbytesleft* shall be 0. If the input
19398 conversion is stopped due to any conditions mentioned above, the value pointed to by *inbytesleft*
19399 shall be non-zero and *errno* shall be set to indicate the condition. If an error occurs *iconv*( ) shall
19400 return (**size_t**)–1 and set *errno* to indicate the error.

19401 **ERRORS**

19402 The *iconv*( ) function shall fail if:

19403 [EILSEQ] Input conversion stopped due to an input byte that does not belong to the
19404 input codeset.

19405 [E2BIG] Input conversion stopped due to lack of space in the output buffer.

19406 [EINVAL] Input conversion stopped due to an incomplete character or shift sequence at
19407 the end of the input buffer.

19408 The *iconv*( ) function may fail if:

19409 [EBADF] The *cd* argument is not a valid open conversion descriptor.

19410 **EXAMPLES**

19411 None.

19412 **APPLICATION USAGE**

19413 The *inbuf* argument indirectly points to the memory area which contains the conversion input
19414 data. The *outbuf* argument indirectly points to the memory area which is to contain the result of
19415 the conversion. The objects indirectly pointed to by *inbuf* and *outbuf* are not restricted to
19416 containing data that is directly representable in the ISO C standard language **char** data type. The
19417 type of *inbuf* and *outbuf*, **char ****, does not imply that the objects pointed to are interpreted as
19418 null-terminated C strings or arrays of characters. Any interpretation of a byte sequence that
19419 represents a character in a given character set encoding scheme is done internally within the
19420 codeset converters. For example, the area pointed to indirectly by *inbuf* and/or *outbuf* can
19421 contain all zero octets that are not interpreted as string terminators but as coded character data
19422 according to the respective codeset encoding scheme. The type of the data (**char**, **short**, **long**, and
19423 so on) read or stored in the objects is not specified, but may be inferred for both the input and
19424 output data by the converters determined by the *fromcode* and *tocode* arguments of *iconv_open*( ).

19425 Regardless of the data type inferred by the converter, the size of the remaining space in both
19426 input and output objects (the *intbytesleft* and *outbytesleft* arguments) is always measured in bytes.

19427 For implementations that support the conversion of state-dependent encodings, the conversion
19428 descriptor must be able to accurately reflect the shift-state in effect at the end of the last
19429 successful conversion. It is not required that the conversion descriptor itself be updated, which
19430 would require it to be a pointer type. Thus, implementations are free to implement the
19431 descriptor as a handle (other than a pointer type) by which the conversion information can be
19432 accessed and updated.

19433 **RATIONALE**

19434 None.

19435 **FUTURE DIRECTIONS**

19436 None.

19437 **SEE ALSO**

19438 *iconv_open*( ), *iconv_close*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<iconv.h>**

19439 **CHANGE HISTORY**

19440 First released in Issue 4. Derived from the HP-UX Manual.

19441 **Issue 6**

19442 The SYNOPSIS has been corrected to align with the **<iconv.h>** reference page.

19443 The **restrict** keyword is added to the *iconv*( ) prototype for alignment with the
19444 ISO/IEC 9899: 1999 standard.

19445 **NAME**

19446 iconv_close — codeset conversion deallocation function

19447 **SYNOPSIS**

19448 XSI
```
#include <iconv.h>
```

19449
```
int iconv_close(iconv_t cd);
```

19450

19451 **DESCRIPTION**

19452 The *iconv_close*( ) function shall deallocate the conversion descriptor *cd* and all other associated |
19453 resources allocated by *iconv_open*( ).

19454 If a file descriptor is used to implement the type **iconv_t**, that file descriptor shall be closed. |

19455 **RETURN VALUE**

19456 Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to
19457 indicate the error.

19458 **ERRORS**

19459 The *iconv_close*( ) function may fail if:

19460 [EBADF] The conversion descriptor is invalid.

19461 **EXAMPLES**

19462 None.

19463 **APPLICATION USAGE**

19464 None.

19465 **RATIONALE**

19466 None.

19467 **FUTURE DIRECTIONS**

19468 None.

19469 **SEE ALSO**

19470 *iconv*( ), *iconv_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<iconv.h>**

19471 **CHANGE HISTORY**

19472 First released in Issue 4. Derived from the HP-UX Manual.

19473 **NAME**

19474 iconv_open — codeset conversion allocation function

19475 **SYNOPSIS**

19476 XSI
```
#include <iconv.h>
```

19477
```
iconv_t iconv_open(const char *tocode, const char *fromcode);
```

19478

19479 **DESCRIPTION**

19480 The *iconv_open*( ) function shall return a conversion descriptor that describes a conversion from
19481 the codeset specified by the string pointed to by the *fromcode* argument to the codeset specified
19482 by the string pointed to by the *tocode* argument. For state-dependent encodings, the conversion
19483 descriptor shall be in a codeset-dependent initial shift state, ready for immediate use with |
19484 *iconv*( ).

19485 Settings of *fromcode* and *tocode* and their permitted combinations are implementation-defined.

19486 A conversion descriptor shall remain valid until it is closed by *iconv_close*( ) or an implicit close. |

19487 If a file descriptor is used to implement conversion descriptors, the FD_CLOEXEC flag shall be
19488 set; see **<fcntl.h>**.

19489 **RETURN VALUE**

19490 Upon successful completion, *iconv_open*( ) shall return a conversion descriptor for use on
19491 subsequent calls to *iconv*( ). Otherwise, *iconv_open*( ) shall return (**iconv_t**)–1 and set *errno* to
19492 indicate the error.

19493 **ERRORS**

19494 The *iconv_open*( ) function may fail if:

19495 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

19496 [ENFILE] Too many files are currently open in the system.

19497 [ENOMEM] Insufficient storage space is available.

19498 [EINVAL] The conversion specified by *fromcode* and *tocode* is not supported by the
19499 implementation.

19500 **EXAMPLES**

19501 None.

19502 **APPLICATION USAGE**

19503 Some implementations of *iconv_open*( ) use *malloc*( ) to allocate space for internal buffer areas.
19504 The *iconv_open*( ) function may fail if there is insufficient storage space to accommodate these
19505 buffers.

19506 Conforming applications must assume that conversion descriptors are not valid after a call to |
19507 one of the *exec* functions.

19508 **RATIONALE**

19509 None.

19510 **FUTURE DIRECTIONS**

19511 None.

19512 **SEE ALSO**

19513 *iconv*( ), *iconv_close*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<iconv.h>**

19514 **CHANGE HISTORY**

19515 First released in Issue 4. Derived from the HP-UX Manual.

19516 **NAME**

19517 if_freenameindex — free memory allocated by *if_nameindex*()

19518 **SYNOPSIS**

19519
```
#include <net/if.h>
```

19520
```
void if_freenameindex(struct if_nameindex *ptr);
```

19521 **DESCRIPTION**

19522 The *if_freenameindex*() function shall free the memory allocated by *if_nameindex*(). The *ptr*
19523 argument shall be a pointer that was returned by *if_nameindex*(). After *if_freenameindex*() has
19524 been called, the application shall not use the array of which *ptr* is the address.

19525 **RETURN VALUE**

19526 None.

19527 **ERRORS**

19528 No errors are defined.

19529 **EXAMPLES**

19530 None.

19531 **APPLICATION USAGE**

19532 None.

19533 **RATIONALE**

19534 None.

19535 **FUTURE DIRECTIONS**

19536 None.

19537 **SEE ALSO**

19538 *getsockopt*(), *if_indextoname*(), *if_nameindex*(), *if_nametoindex*(), *setsockopt*(), the Base Definitions
19539 volume of IEEE Std 1003.1-200x, **<net/if.h>**

19540 **CHANGE HISTORY**

19541 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19542 **NAME**

19543 if_indextoname — map a network interface index to its corresponding name

19544 **SYNOPSIS**

19545 `#include <net/if.h>`

19546 `char *if_indextoname(unsigned ifindex, char *ifname);`

19547 **DESCRIPTION**

19548 The *if_indextoname*( ) function shall map an interface index to its corresponding name.

19549 When this function is called, *ifname* shall point to a buffer of at least {IFNAMSIZ} bytes. The
19550 function shall place in this buffer the name of the interface with index *ifindex*.

19551 **RETURN VALUE**

19552 If *ifindex* is an interface index, then the function shall return the value supplied in *ifname*, which
19553 points to a buffer now containing the interface name. Otherwise, the function shall return a
19554 NULL pointer and set *errno* to indicate the error.

19555 **ERRORS**

19556 The *if_indextoname*( ) function shall fail if:

19557 [ENXIO] The interface does not exist.

19558 **EXAMPLES**

19559 None.

19560 **APPLICATION USAGE**

19561 None.

19562 **RATIONALE**

19563 None.

19564 **FUTURE DIRECTIONS**

19565 None.

19566 **SEE ALSO**

19567 *getsockopt*( ), *if_freenameindex*( ), *if_nameindex*( ), *if_nametoindex*( ), *setsockopt*( ), the Base
19568 Definitions volume of IEEE Std 1003.1-200x, **<net/if.h>**

19569 **CHANGE HISTORY**

19570 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19571 **NAME**

19572 if_nameindex — return all network interface names and indexes

19573 **SYNOPSIS**

19574 `#include <net/if.h>`

19575 `struct if_nameindex *`*`if_nameindex`*`(void);`

19576 **DESCRIPTION**

19577 The *if_nameindex*( ) function shall return an array of *if_nameindex* structures, one structure per
19578 interface. The end of the array is indicated by a structure with an *if_index* field of zero and an
19579 *if_name* field of NULL.

19580 Applications should call *if_freenameindex*( ) to release the memory that may be dynamically
19581 allocated by this function, after they have finished using it.

19582 **RETURN VALUE**

19583 Array of structures identifying local interfaces. A NULL pointer is returned upon an error, with
19584 *errno* set to indicate the error.

19585 **ERRORS**

19586 The *if_nameindex*( ) function may fail if:

19587 [ENOBUFS] Insufficient resources are available to complete the function.

19588 **EXAMPLES**

19589 None.

19590 **APPLICATION USAGE**

19591 None.

19592 **RATIONALE**

19593 None.

19594 **FUTURE DIRECTIONS**

19595 None.

19596 **SEE ALSO**

19597 *getsockopt*( ), *if_freenameindex*( ), *if_indextoname*( ), *if_nametoindex*( ), *setsockopt*( ), the Base
19598 Definitions volume of IEEE Std 1003.1-200x, **<net/if.h>**

19599 **CHANGE HISTORY**

19600 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19601 **NAME**

19602 if_nametoindex — map a network interface name to its corresponding index

19603 **SYNOPSIS**

19604 `#include <net/if.h>`

19605 `unsigned if_nametoindex(const char *ifname);`

19606 **DESCRIPTION**

19607 The *if_nametoindex*( ) function shall return the interface index corresponding to name *ifname*.

19608 **RETURN VALUE**

19609 The corresponding index if *ifname* is the name of an interface; otherwise, zero.

19610 **ERRORS**

19611 No errors are defined.

19612 **EXAMPLES**

19613 None.

19614 **APPLICATION USAGE**

19615 None.

19616 **RATIONALE**

19617 None.

19618 **FUTURE DIRECTIONS**

19619 None.

19620 **SEE ALSO**

19621 *getsockopt*( ), *if_freenameindex*( ), *if_indextoname*( ), *if_nameindex*( ), *setsockopt*( ), the Base
19622 Definitions volume of IEEE Std 1003.1-200x, **<net/if.h>**

19623 **CHANGE HISTORY**

19624 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19625 **NAME**

19626 ilogb, ilogbf, ilogbl — return an unbiased exponent

19627 **SYNOPSIS**

19628
```
#include <math.h>
```

19629
```
int ilogb(double x);
```
19630
```
int ilogbf(float x);
```
19631
```
int ilogbl(long double x);
```

19632 **DESCRIPTION**

19633 CX The functionality described on this reference page is aligned with the ISO C standard. Any
19634 conflict between the requirements described here and the ISO C standard is unintentional. This
19635 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

19636 These functions shall return the exponent part of their argument *x*. Formally, the return value is
19637 the integral part of $\log_r |x|$ as a signed integral value, for non-zero *x*, where *r* is the radix of the
19638 machine's floating-point arithmetic, which is the value of FLT_RADIX defined in **<float.h>**.

19639 An application wishing to check for error situations should set *errno* to zero and call
19640 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
19641 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
19642 zero, an error has occurred.

19643 **RETURN VALUE**

19644 Upon successful completion, these functions shall return the exponent part of *x* as a signed
19645 integer value. They are equivalent to calling the corresponding *logb*( ) function and casting the
19646 returned value to type **int**.

19647 XSI If *x* is 0, a domain error shall occur, and the value FP_ILOGB0 shall be returned.

19648 XSI If *x* is ±Inf, a domain error shall occur, and the value {INT_MAX} shall be returned.

19649 XSI If *x* is a NaN, a domain error shall occur, and the value FP_ILOGBNAN shall be returned.

19650 XSI If the correct value is greater than {INT_MAX}, {INT_MAX} shall be returned and a domain error |
19651 shall occur.

19652 If the correct value is less than {INT_MIN}, {INT_MIN} shall be returned and a domain error |
19653 shall occur.

19654 **ERRORS**

19655 These functions shall fail if:

19656 XSI	Domain Error	The *x* argument is zero, NaN, or ±Inf, or the correct value is not representable as an integer.
19657		
19658		If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
19659		
19660		
19661		

19662 **EXAMPLES**

19663 None.

19664 **APPLICATION USAGE**

19665 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
19666 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

19667 **RATIONALE**

19668 The errors come from taking the expected floating-point value and converting it to **int**, which is
19669 invalid operation in IEEE Std 754-1985 (since overflow, infinity, and NaN are not representable
19670 in a type **int**), so should be a domain error.

19671 There are no known implementations that overflow. For overflow to happen, {INT_MAX} must
19672 be less than LDBL_MAX_EXP**log2*(FLT_RADIX) or {INT_MIN} must be greater than
19673 LDBL_MIN_EXP**log2*(FLT_RADIX) if subnormals are not supported, or {INT_MIN} must be
19674 greater than (LDBL_MIN_EXP-LDBL_MANT_DIG)**log2*(FLT_RADIX) if subnormals are
19675 supported.

19676 **FUTURE DIRECTIONS**

19677 None.

19678 **SEE ALSO**

19679 *feclearexcept*( ), *fetestexcept*( ), *logb*( ), *scalb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, |
19680 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<float.h>**, **<math.h>** |

19681 **CHANGE HISTORY**

19682 First released in Issue 4, Version 2.

19683 **Issue 5**

19684 Moved from X/OPEN UNIX extension to BASE.

19685 **Issue 6**

19686 The *ilogb*( ) function is no longer marked as an extension.

19687 The *ilogbf*( ) and *ilogbl*( ) functions are added for alignment with the ISO/IEC 9899:1999
19688 standard.

19689 The RETURN VALUE section is revised for alignment with the ISO/IEC 9899:1999 standard.

19690 XSI extensions are marked.

19691 **NAME**

19692 imaxabs — return absolute value

19693 **SYNOPSIS**

19694
```
#include <inttypes.h>
```

19695
```
intmax_t imaxabs(intmax_t j);
```

19696 **DESCRIPTION**

19697 CX The functionality described on this reference page is aligned with the ISO C standard. Any
19698 conflict between the requirements described here and the ISO C standard is unintentional. This
19699 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

19700 The *imaxabs*( ) function shall compute the absolute value of an integer *j*. If the result cannot be
19701 represented, the behavior is undefined.

19702 **RETURN VALUE**

19703 The *imaxabs*( ) function shall return the absolute value.

19704 **ERRORS**

19705 No errors are defined.

19706 **EXAMPLES**

19707 None.

19708 **APPLICATION USAGE**

19709 The absolute value of the most negative number cannot be represented in two's complement.

19710 **RATIONALE**

19711 None.

19712 **FUTURE DIRECTIONS**

19713 None.

19714 **SEE ALSO**

19715 *imaxdiv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<inttypes.h>**

19716 **CHANGE HISTORY**

19717 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

19718 **NAME**

19719 imaxdiv — return quotient and remainder

19720 **SYNOPSIS**

19721 `#include <inttypes.h>`

19722 `imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);`

19723 **DESCRIPTION**

19724 CX The functionality described on this reference page is aligned with the ISO C standard. Any
19725 conflict between the requirements described here and the ISO C standard is unintentional. This
19726 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

19727 The *imaxdiv*( ) function shall compute *numer* / *denom* and *numer* % *denom* in a single operation.

19728 **RETURN VALUE**

19729 The *imaxdiv*( ) function shall return a structure of type **imaxdiv_t**, comprising both the quotient
19730 and the remainder. The structure shall contain (in either order) the members *quot* (the quotient)
19731 and *rem* (the remainder), each of which has type **intmax_t**.

19732 If either part of the result cannot be represented, the behavior is undefined.

19733 **ERRORS**

19734 No errors are defined.

19735 **EXAMPLES**

19736 None.

19737 **APPLICATION USAGE**

19738 None.

19739 **RATIONALE**

19740 None.

19741 **FUTURE DIRECTIONS**

19742 None.

19743 **SEE ALSO**

19744 *imaxabs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<inttypes.h>**

19745 **CHANGE HISTORY**

19746 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

19747 **NAME**

19748 index — character string operations (**LEGACY**)

19749 **SYNOPSIS**

19750 XSI
```
#include <strings.h>

char *index(const char *s, int c);
```
19751

19752

19753 **DESCRIPTION**

19754 The *index*( ) function shall be equivalent to *strchr*( ).

19755 **RETURN VALUE**

19756 See *strchr*( ).

19757 **ERRORS**

19758 See *strchr*( ).

19759 **EXAMPLES**

19760 None.

19761 **APPLICATION USAGE**

19762 *strchr*( ) is preferred over this function.

19763 For maximum portability, it is recommended to replace the function call to *index*( ) as follows:

19764
```
#define index(a,b) strchr((a),(b))
```
19765 **RATIONALE**

19766 None.

19767 **FUTURE DIRECTIONS**

19768 This function may be withdrawn in a future version.

19769 **SEE ALSO**

19770 *strchr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

19771 **CHANGE HISTORY**

19772 First released in Issue 4, Version 2.

19773 **Issue 5**

19774 Moved from X/OPEN UNIX extension to BASE.

19775 **Issue 6**

19776 This function is marked LEGACY.

19777 **NAME**

19778 inet_addr, inet_ntoa — IPv4 address manipulation

19779 **SYNOPSIS**

19780 `#include <arpa/inet.h>`

19781 `in_addr_t inet_addr(const char *cp);`
19782 `char *inet_ntoa(struct in_addr in);`

19783 **DESCRIPTION**

19784 The *inet_addr*( ) function shall convert the string pointed to by *cp*, in the standard IPv4 dotted
19785 decimal notation, to an integer value suitable for use as an Internet address.

19786 The *inet_ntoa*( ) function shall convert the Internet host address specified by *in* to a string in the
19787 Internet standard dot notation.

19788 The *inet_ntoa*( ) function need not be reentrant. A function that is not required to be reentrant is
19789 not required to be thread-safe.

19790 All Internet addresses shall be returned in network order (bytes ordered from left to right).

19791 Values specified using IPv4 dotted decimal notation take one of the following forms:

19792 `a.b.c.d` When four parts are specified, each shall be interpreted as a byte of data and |
19793 assigned, from left to right, to the four bytes of an Internet address. |

19794 `a.b.c` When a three-part address is specified, the last part shall be interpreted as a 16-bit |
19795 quantity and placed in the rightmost two bytes of the network address. This makes |
19796 the three-part address format convenient for specifying Class B network addresses |
19797 as **128.net.host**. |

19798 `a.b` When a two-part address is supplied, the last part shall be interpreted as a 24-bit |
19799 quantity and placed in the rightmost three bytes of the network address. This |
19800 makes the two-part address format convenient for specifying Class A network |
19801 addresses as **net.host**.

19802 `a` When only one part is given, the value shall be stored directly in the network |
19803 address without any byte rearrangement.

19804 All numbers supplied as parts in IPv4 dotted decimal notation may be decimal, octal, or
19805 hexadecimal, as specified in the ISO C standard (that is, a leading 0x or 0X implies hexadecimal;
19806 otherwise, a leading `'0'` implies octal; otherwise, the number is interpreted as decimal).

19807 **RETURN VALUE**

19808 Upon successful completion, *inet_addr*( ) shall return the Internet address. Otherwise, it shall
19809 return (**in_addr_t**)(–1).

19810 The *inet_ntoa*( ) function shall return a pointer to the network address in Internet standard dot
19811 notation.

19812 **ERRORS**

19813 No errors are defined.

19814 **EXAMPLES**

19815 None.

19816 **APPLICATION USAGE**

19817 The return value of *inet_ntoa*( ) may point to static data that may be overwritten by subsequent
19818 calls to *inet_ntoa*( ).

19819 **RATIONALE**

19820 None.

19821 **FUTURE DIRECTIONS**

19822 None.

19823 **SEE ALSO**

19824 *endhostent*( ), *endnetent*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<arpa/inet.h>**

19825 **CHANGE HISTORY**

19826 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19827 **NAME**

19828 inet_ntoa — IPv4 address manipulation

19829 **SYNOPSIS**

19830 `#include <arpa/inet.h>`

19831 `char *inet_ntoa(struct in_addr in);`

19832 **DESCRIPTION**

19833 Refer to *inet_addr*( ).

19834 **NAME**

19835 inet_ntop, inet_pton — convert IPv4 and IPv6 addresses between binary and text form

19836 **SYNOPSIS**

19837
```
#include <arpa/inet.h>
```

19838
```
const char *inet_ntop(int af, const void *restrict src,
    char *restrict dst, socklen_t size);
int inet_pton(int af, const char *restrict src, void *restrict dst);
```

19841 **DESCRIPTION**

19842 The *inet_ntop*() function shall convert a numeric address into a text string suitable for
19843 IP6 presentation. The *af* argument shall specify the family of the address. This can be AF_INET or
19844 AF_INET6. The *src* argument points to a buffer holding an IPv4 address if the *af* argument is
19845 IP6 AF_INET, or an IPv6 address if the *af* argument is AF_INET6. The *dst* argument points to a
19846 buffer where the function stores the resulting text string; it shall not be NULL. The *size* argument
19847 specifies the size of this buffer, which shall be large enough to hold the text string
19848 IP6 (INET_ADDRSTRLEN characters for IPv4, INET6_ADDRSTRLEN characters for IPv6).

19849 The *inet_pton*() function shall convert an address in its standard text presentation form into its
19850 IP6 numeric binary form. The *af* argument shall specify the family of the address. The AF_INET and
19851 AF_INET6 address families shall be supported. The *src* argument points to the string being
19852 passed in. The *dst* argument points to a buffer into which the function stores the numeric
19853 IP6 address; this shall be large enough to hold the numeric address (32 bits for AF_INET, 128 bits for
19854 AF_INET6).

19855 If the *af* argument of *inet_pton*() is AF_INET, the *src* string shall be in the standard IPv4 dotted-
19856 decimal form:

19857
```
ddd.ddd.ddd.ddd
```

19858 where "ddd" is a one to three digit decimal number between 0 and 255 (see *inet_addr*()). The
19859 *inet_pton*() function does not accept other formats (such as the octal numbers, hexadecimal
19860 numbers, and fewer than four numbers that *inet_addr*() accepts).

19861 IP6 If the *af* argument of *inet_pton*() is AF_INET6, the *src* string shall be in one of the following
19862 standard IPv6 text forms:

19863 1. The preferred form is "x:x:x:x:x:x:x:x", where the 'x's are the hexadecimal values
19864 of the eight 16-bit pieces of the address. Leading zeros in individual fields can be omitted,
19865 but there shall be at least one numeral in every field.

19866 2. A string of contiguous zero fields in the preferred form can be shown as "::". The "::"
19867 can only appear once in an address. Unspecified addresses ("0:0:0:0:0:0:0:0") may
19868 be represented simply as "::".

19869 3. A third form that is sometimes more convenient when dealing with a mixed environment
19870 of IPv4 and IPv6 nodes is "x:x:x:x:x:x:d.d.d.d", where the 'x's are the
19871 hexadecimal values of the six high-order 16-bit pieces of the address, and the 'd's are the
19872 decimal values of the four low-order 8-bit pieces of the address (standard IPv4
19873 representation).

19874 **Note:** A more extensive description of the standard representations of IPv6 addresses can be found in
19875 RFC 2373.

19876

19877 **RETURN VALUE**

19878 The *inet_ntop*( ) function shall return a pointer to the buffer containing the text string if the
19879 conversion succeeds, and NULL otherwise, and set *errno* to indicate the error.

19880 The *inet_pton*( ) function shall return 1 if the conversion succeeds, with the address pointed to by
19881 IP6 *dst* in network byte order. It shall return 0 if the input is not a valid IPv4 dotted-decimal string or
19882 a valid IPv6 address string, or –1 with *errno* set to [EAFNOSUPPORT] if the *af* argument is
19883 unknown.

19884 **ERRORS**

19885 The *inet_ntop*( ) and *inet_pton*( ) functions shall fail if:

19886 [EAFNOSUPPORT]
19887 The *af* argument is invalid.

19888 [ENOSPC] The size of the *inet_ntop*( ) result buffer is inadequate.

19889 **EXAMPLES**

19890 None.

19891 **APPLICATION USAGE**

19892 None.

19893 **RATIONALE**

19894 None.

19895 **FUTURE DIRECTIONS**

19896 None.

19897 **SEE ALSO**

19898 The Base Definitions volume of IEEE Std 1003.1-200x, **<arpa/inet.h>**

19899 **CHANGE HISTORY**

19900 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19901 IPv6 extensions are marked.

19902 The **restrict** keyword is added to the *inet_ntop*( ) and *inet_pton*( ) prototypes for alignment with
19903 the ISO / IEC 9899: 1999 standard.

19904 **NAME**

19905 initstate, random, setstate, srandom — pseudo-random number functions

19906 **SYNOPSIS**

19907 XSI
```
#include <stdlib.h>

char *initstate(unsigned seed, char *state, size_t size);
long random(void);
char *setstate(const char *state);
void srandom(unsigned seed);
```
19912

19913 **DESCRIPTION**

19914 The *random*( ) function shall use a non-linear additive feedback random-number generator |
19915 employing a default state array size of 31 **long** integers to return successive pseudo-random |
19916 numbers in the range from 0 to 2^{31}–1. The period of this random-number generator is
19917 approximately 16 x (2^{31}–1). The size of the state array determines the period of the random-
19918 number generator. Increasing the state array size shall increase the period. |

19919 With 256 bytes of state information, the period of the random-number generator shall be greater |
19920 than 2^{69}. |

19921 Like *rand*( ), *random*( ) shall produce by default a sequence of numbers that can be duplicated by |
19922 calling *srandom*( ) with 1 as the seed. |

19923 The *srandom*( ) function shall initialize the current state array using the value of *seed*. |

19924 The *initstate*( ) and *setstate*( ) functions handle restarting and changing random-number
19925 generators. The *initstate*( ) function allows a state array, pointed to by the *state* argument, to be
19926 initialized for future use. The *size* argument, which specifies the size in bytes of the state array, |
19927 shall be used by *initstate*( ) to decide what type of random-number generator to use; the larger |
19928 the state array, the more random the numbers. Values for the amount of state information are 8,
19929 32, 64, 128, and 256 bytes. Other values greater than 8 bytes are rounded down to the nearest one
19930 of these values. If *initstate*( ) is called with 8≤*size*<32, then *random*( ) shall use a simple linear |
19931 congruential random number generator. The *seed* argument specifies a starting point for the |
19932 random-number sequence and provides for restarting at the same point. The *initstate*( ) function
19933 shall return a pointer to the previous state information array.

19934 If *initstate*( ) has not been called, then *random*( ) shall behave as though *initstate*( ) had been called |
19935 with *seed*=1 and *size*=128.

19936 Once a state has been initialized, *setstate*( ) allows switching between state arrays. The array
19937 defined by the *state* argument shall be used for further random-number generation until |
19938 *initstate*( ) is called or *setstate*( ) is called again. The *setstate*( ) function shall return a pointer to the
19939 previous state array.

19940 **RETURN VALUE**

19941 If *initstate*( ) is called with *size* less than 8, it shall return NULL.

19942 The *random*( ) function shall return the generated pseudo-random number.

19943 The *srandom*( ) function shall not return a value.

19944 Upon successful completion, *initstate*( ) and *setstate*( ) shall return a pointer to the previous state
19945 array; otherwise, a null pointer shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

After initialization, a state array can be restarted at a different point in one of two ways:

1. The *initstate*( ) function can be used, with the desired seed, state array, and size of the array.
2. The *setstate*( ) function, with the desired state, can be used, followed by *srandom*( ) with the desired seed. The advantage of using both of these functions is that the size of the state array does not have to be saved once it is initialized.

Although some implementations of *random*( ) have written messages to standard error, such implementations do not conform to this volume of IEEE Std 1003.1-200x.

Issue 5 restores the historical behavior of this function.

Threaded applications should use *rand_r*( ), *erand48*( ), *nrand48*( ), or *jrand48*( ) instead of *random*( ) when an independent random number sequence in multiple threads is required.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

drand48( ), *rand*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

In the DESCRIPTION, the phrase ‘‘values smaller than 8’’ is replaced with ‘‘values greater than or equal to 8, or less than 32’’, ‘‘*size*<8’’ is replaced with ‘‘8≤*size* <32’’, and a new first paragraph is added to the RETURN VALUE section. A note is added to the APPLICATION USAGE indicating that these changes restore the historical behavior of the function.

Issue 6

In the DESCRIPTION, duplicate text ‘‘For values greater than or equal to 8 . . .’’ is removed.

NAME

insque, remque — insert or remove an element in a queue

SYNOPSIS

XSI
```
#include <search.h>

void insque(void *element, void *pred);
void remque(void *element);
```

DESCRIPTION

The *insque*( ) and *remque*( ) functions shall manipulate queues built from doubly-linked lists. The queue can be either circular or linear. An application using *insque*( ) or *remque*( ) shall ensure it defines a structure in which the first two members of the structure are pointers to the same type of structure, and any further members are application-specific. The first member of the structure is a forward pointer to the next entry in the queue. The second member is a backward pointer to the previous entry in the queue. If the queue is linear, the queue is terminated with null pointers. The names of the structure and of the pointer members are not subject to any special restriction.

The *insque*( ) function shall insert the element pointed to by *element* into a queue immediately after the element pointed to by *pred*.

The *remque*( ) function shall remove the element pointed to by *element* from a queue.

If the queue is to be used as a linear list, invoking *insque*(&*element*, NULL), where *element* is the initial element of the queue, shall initialize the forward and backward pointers of *element* to null pointers.

If the queue is to be used as a circular list, the application shall ensure it initializes the forward pointer and the backward pointer of the initial element of the queue to the element's own address.

RETURN VALUE

The *insque*( ) and *remque*( ) functions do not return a value.

ERRORS

No errors are defined.

EXAMPLES

Creating a Linear Linked List

The following example creates a linear linked list.

```
#include <search.h>
...
struct myque element1;
struct myque element2;

char *data1 = "DATA1";
char *data2 = "DATA2";
...
element1.data = data1;
element2.data = data2;

insque (&element1, NULL);
insque (&element2, &element1);
```

20021 **Creating a Circular Linked List**

20022 The following example creates a circular linked list.

```
20023 #include <search.h>
20024 ...
20025 struct myque element1;
20026 struct myque element2;
20027 char *data1 = "DATA1";
20028 char *data2 = "DATA2";
20029 ...
20030 element1.data = data1;
20031 element2.data = data2;
20032 element1.fwd = &element1;
20033 element1.bck = &element1;
20034 insque (&element2, &element1);
```

20035 **Removing an Element**

20036 The following example removes the element pointed to by *element1*.

```
20037 #include <search.h>
20038 ...
20039 struct myque element1;
20040 ...
20041 remque (&element1);
```

20042 **APPLICATION USAGE**

20043 The historical implementations of these functions described the arguments as being of type
20044 **struct qelem *** rather than as being of type **void *** as defined here. In those implementations,
20045 **struct qelem** was commonly defined in **<search.h>** as:

```
20046 struct qelem {
20047     struct qelem  *q_forw;
20048     struct qelem  *q_back;
20049 };
```

20050 Applications using these functions, however, were never able to use this structure directly since
20051 it provided no room for the actual data contained in the elements. Most applications defined
20052 structures that contained the two pointers as the initial elements and also provided space for, or
20053 pointers to, the object's data. Applications that used these functions to update more than one
20054 type of table also had the problem of specifying two or more different structures with the same
20055 name, if they literally used **struct qelem** as specified.

20056 As described here, the implementations were actually expecting a structure type where the first
20057 two members were forward and backward pointers to structures. With C compilers that didn't
20058 provide function prototypes, applications used structures as specified in the DESCRIPTION
20059 above and the compiler did what the application expected.

20060 If this method had been carried forward with an ISO C standard compiler and the historical
20061 function prototype, most applications would have to be modified to cast pointers to the
20062 structures actually used to be pointers to **struct qelem** to avoid compilation warnings. By
20063 specifying **void *** as the argument type, applications do not need to change (unless they
20064 specifically referenced **struct qelem** and depended on it being defined in **<search.h>**).

20065 **RATIONALE**

20066 None.

20067 **FUTURE DIRECTIONS**

20068 None.

20069 **SEE ALSO**

20070 The Base Definitions volume of IEEE Std 1003.1-200x, **<search.h>**

20071 **CHANGE HISTORY**

20072 First released in Issue 4, Version 2.

20073 **Issue 5**

20074 Moved from X/OPEN UNIX extension to BASE.

20075 **Issue 6**

20076 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

20077 **NAME**

20078 ioctl — control a STREAMS device (**STREAMS**)

20079 **SYNOPSIS**

20080 XSR
```
#include <stropts.h>

int ioctl(int fildes, int request, ... /* arg */);
```
20081

20082

20083 **DESCRIPTION**

20084 The *ioctl*( ) function shall perform a variety of control functions on STREAMS devices. For non-
20085 STREAMS devices, the functions performed by this call are unspecified. The *request* argument
20086 and an optional third argument (with varying type) shall be passed to and interpreted by the
20087 appropriate part of the STREAM associated with *fildes*.

20088 The *fildes* argument is an open file descriptor that refers to a device.

20089 The *request* argument selects the control function to be performed and shall depend on the
20090 STREAMS device being addressed.

20091 The *arg* argument represents additional information that is needed by this specific STREAMS
20092 device to perform the requested function. The type of *arg* depends upon the particular control
20093 request, but it shall be either an integer or a pointer to a device-specific data structure.

20094 The *ioctl*( ) commands applicable to STREAMS, their arguments, and error conditions that apply
20095 to each individual command are described below.

20096 The following *ioctl*( ) commands, with error values indicated, are applicable to all STREAMS
20097 files:

20098 I_PUSH Pushes the module whose name is pointed to by *arg* onto the top of the
20099 current STREAM, just below the STREAM head. It then calls the *open*( )
20100 function of the newly-pushed module.

20101 The *ioctl*( ) function with the I_PUSH command shall fail if:

20102 [EINVAL] Invalid module name.

20103 [ENXIO] Open function of new module failed.

20104 [ENXIO] Hangup received on *fildes*.

20105 I_POP Removes the module just below the STREAM head of the STREAM pointed to
20106 by *fildes*. The *arg* argument should be 0 in an I_POP request.

20107 The *ioctl*( ) function with the I_POP command shall fail if:

20108 [EINVAL] No module present in the STREAM.

20109 [ENXIO] Hangup received on *fildes*.

20110 I_LOOK Retrieves the name of the module just below the STREAM head of the
20111 STREAM pointed to by *fildes*, and places it in a character string pointed to by
20112 *arg*. The buffer pointed to by *arg* should be at least FMNAMESZ+1 bytes long,
20113 where FMNAMESZ is defined in **<stropts.h>**.

20114 The *ioctl*( ) function with the I_LOOK command shall fail if:

20115 [EINVAL] No module present in the STREAM.

20116 I_FLUSH Flushes read and/or write queues, depending on the value of *arg*. Valid *arg*
20117 values are:

FLUSHR Flush all read queues.

FLUSHW Flush all write queues.

FLUSHRW Flush all read and all write queues.

The *ioctl*( ) function with the I_FLUSH command shall fail if:

[EINVAL] Invalid *arg* value.

[EAGAIN] or [ENOSR]
Unable to allocate buffers for flush message.

[ENXIO] Hangup received on *fildes*.

I_FLUSHBAND Flushes a particular band of messages. The *arg* argument points to a **bandinfo** structure. The *bi_flag* member may be one of FLUSHR, FLUSHW, or FLUSHRW as described above. The *bi_pri* member determines the priority band to be flushed.

I_SETSIG Requests that the STREAMS implementation send the SIGPOLL signal to the calling process when a particular event has occurred on the STREAM associated with *fildes*. I_SETSIG supports an asynchronous processing capability in STREAMS. The value of *arg* is a bitmask that specifies the events for which the process should be signaled. It is the bitwise-inclusive OR of any combination of the following constants:

S_RDNORM A normal (priority band set to 0) message has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.

S_RDBAND A message with a non-zero priority band has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.

S_INPUT A message, other than a high-priority message, has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.

S_HIPRI A high-priority message is present on a STREAM head read queue. A signal shall be generated even if the message is of zero length.

S_OUTPUT The write queue for normal data (priority band 0) just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) normal data downstream.

S_WRNORM Equivalent to S_OUTPUT. |

S_WRBAND The write queue for a non-zero priority band just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) priority data downstream.

S_MSG A STREAMS signal message that contains the SIGPOLL signal has reached the front of the STREAM head read queue.

S_ERROR Notification of an error condition has reached the STREAM head.

S_HANGUP Notification of a hangup has reached the STREAM head.

S_BANDURG When used in conjunction with S_RDBAND, SIGURG is generated instead of SIGPOLL when a priority message reaches the front of the STREAM head read queue.

If *arg* is 0, the calling process shall be unregistered and shall not receive further SIGPOLL signals for the stream associated with *fildes*.

Processes that wish to receive SIGPOLL signals shall ensure that they explicitly register to receive them using I_SETSIG. If several processes register to receive this signal for the same event on the same STREAM, each process shall be signaled when the event occurs.

The *ioctl*( ) function with the I_SETSIG command shall fail if:

[EINVAL] The value of *arg* is invalid.

[EINVAL] The value of *arg* is 0 and the calling process is not registered to receive the SIGPOLL signal.

[EAGAIN] There were insufficient resources to store the signal request.

I_GETSIG Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal. The events are returned as a bitmask in an **int** pointed to by *arg*, where the events are those specified in the description of I_SETSIG above.

The *ioctl*( ) function with the I_GETSIG command shall fail if:

[EINVAL] Process is not registered to receive the SIGPOLL signal.

I_FIND Compares the names of all modules currently present in the STREAM to the name pointed to by *arg*, and returns 1 if the named module is present in the STREAM, or returns 0 if the named module is not present.

The *ioctl*( ) function with the I_FIND command shall fail if:

[EINVAL] *arg* does not contain a valid module name.

I_PEEK Retrieves the information in the first message on the STREAM head read queue without taking the message off the queue. It is analogous to *getmsg*( ) except that this command does not remove the message from the queue. The *arg* argument points to a **strpeek** structure.

The application shall ensure that the *maxlen* member in the **ctlbuf** and **databuf** **strbuf** structures is set to the number of bytes of control information and/or data information, respectively, to retrieve. The *flags* member may be marked RS_HIPRI or 0, as described by *getmsg*( ). If the process sets *flags* to RS_HIPRI, for example, I_PEEK shall only look for a high-priority message on the STREAM head read queue.

I_PEEK returns 1 if a message was retrieved, and returns 0 if no message was found on the STREAM head read queue, or if the RS_HIPRI flag was set in *flags* and a high-priority message was not present on the STREAM head read queue. It does not wait for a message to arrive. On return, **ctlbuf** specifies information in the control buffer, **databuf** specifies information in the data buffer, and *flags* contains the value RS_HIPRI or 0.

I_SRDOPT Sets the read mode using the value of the argument *arg*. Read modes are described in *read*( ). Valid *arg* flags are:

RNORM Byte-stream mode, the default.

RMSGD Message-discard mode.

RMSGN Message-nondiscard mode.

The bitwise-inclusive OR of RMSGD and RMSGN shall return [EINVAL]. The bitwise-inclusive OR of RNORM and either RMSGD or RMSGN shall result in the other flag overriding RNORM which is the default.

In addition, treatment of control messages by the STREAM head may be changed by setting any of the following flags in *arg*:

RPROTNORM Fail *read*( ) with [EBADMSG] if a message containing a control part is at the front of the STREAM head read queue.

RPROTDAT Deliver the control part of a message as data when a process issues a *read*( ).

RPROTDIS Discard the control part of a message, delivering any data portion, when a process issues a *read*( ).

The *ioctl*( ) function with the I_SRDOPT command shall fail if:

[EINVAL] The *arg* argument is not valid.

I_GRDOPT Returns the current read mode setting as, described above, in an **int** pointed to by the argument *arg*. Read modes are described in *read*( ).

I_NREAD Counts the number of data bytes in the data part of the first message on the STREAM head read queue and places this value in the **int** pointed to by *arg*. The return value for the command shall be the number of messages on the STREAM head read queue. For example, if 0 is returned in *arg*, but the *ioctl*( ) return value is greater than 0, this indicates that a zero-length message is next on the queue.

I_FDINSERT Creates a message from specified buffer(s), adds information about another STREAM, and sends the message downstream. The message contains a control part and an optional data part. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The *arg* argument points to a **strfdinsert** structure.

The application shall ensure that the *len* member in the **ctlbuf strbuf** structure is set to the size of a **t_uscalar_t** plus the number of bytes of control information to be sent with the message. The *fildes* member specifies the file descriptor of the other STREAM, and the *offset* member, which must be suitably aligned for use as a **t_uscalar_t**, specifies the offset from the start of the control buffer where I_FDINSERT shall store a **t_uscalar_t** whose interpretation is specific to the STREAM end. The application shall ensure that the *len* member in the **databuf strbuf** structure is set to the number of bytes of data information to be sent with the message, or to 0 if no data part is to be sent.

The *flags* member specifies the type of message to be created. A normal message is created if *flags* is set to 0, and a high-priority message is created if *flags* is set to RS_HIPRI. For non-priority messages, I_FDINSERT shall block if the STREAM write queue is full due to internal flow control conditions. For priority messages, I_FDINSERT does not block on this condition. For non-priority messages, I_FDINSERT does not block when the write queue is full

and O_NONBLOCK is set. Instead, it fails and sets *errno* to [EAGAIN].

I_FDINSERT also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the STREAM, regardless of priority or whether O_NONBLOCK has been specified. No partial message is sent.

The *ioctl*( ) function with the I_FDINSERT command shall fail if:

[EAGAIN] A non-priority message is specified, the O_NONBLOCK flag is set, and the STREAM write queue is full due to internal flow control conditions.

[EAGAIN] or [ENOSR]
Buffers cannot be allocated for the message that is to be created.

[EINVAL] One of the following:

— The *fildes* member of the **strfdinsert** structure is not a valid, open STREAM file descriptor.

— The size of a **t_uscalar_t** plus *offset* is greater than the *len* member for the buffer specified through **ctlbuf**.

— The *offset* member does not specify a properly-aligned location in the data buffer.

— An undefined value is stored in *flags*.

[ENXIO] Hangup received on the STREAM identified by either the *fildes* argument or the *fildes* member of the **strfdinsert** structure.

[ERANGE] The *len* member for the buffer specified through **databuf** does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAM module or the *len* member for the buffer specified through **databuf** is larger than the maximum configured size of the data part of a message; or the *len* member for the buffer specified through **ctlbuf** is larger than the maximum configured size of the control part of a message.

I_STR Constructs an internal STREAMS *ioctl*( ) message from the data pointed to by *arg*, and sends that message downstream.

This mechanism is provided to send *ioctl*( ) requests to downstream modules and drivers. It allows information to be sent with *ioctl*( ), and returns to the process any information sent upstream by the downstream recipient. I_STR shall block until the system responds with either a positive or negative acknowledgement message, or until the request times out after some period of time. If the request times out, it shall fail with *errno* set to [ETIME].

At most, one I_STR can be active on a STREAM. Further I_STR calls shall block until the active I_STR completes at the STREAM head. The default timeout interval for these requests is 15 seconds. The O_NONBLOCK flag has no effect on this call.

To send requests downstream, the application shall ensure that *arg* points to a **strioctl** structure.

The *ic_cmd* member is the internal *ioctl*() command intended for a downstream module or driver and *ic_timout* is the number of seconds (–1=infinite, 0=use implementation-defined timeout interval, >0=as specified) an I_STR request shall wait for acknowledgement before timing out. *ic_len* is the number of bytes in the data argument, and *ic_dp* is a pointer to the data argument. The *ic_len* member has two uses: on input, it contains the length of the data argument passed in, and on return from the command, it contains the number of bytes being returned to the process (the buffer pointed to by *ic_dp* should be large enough to contain the maximum amount of data that any module or the driver in the STREAM can return).

The STREAM head shall convert the information pointed to by the **strioctl** structure to an internal *ioctl*() command message and sends it downstream.

The *ioctl*() function with the I_STR command shall fail if:

[EAGAIN] or [ENOSR]
: Unable to allocate buffers for the *ioctl*() message.

[EINVAL]
: The *ic_len* member is less than 0 or larger than the maximum configured size of the data part of a message, or *ic_timout* is less than –1.

[ENXIO]
: Hangup received on *fildes*.

[ETIME]
: A downstream *ioctl*() timed out before acknowledgement was received.

An I_STR can also fail while waiting for an acknowledgement if a message indicating an error or a hangup is received at the STREAM head. In addition, an error code can be returned in the positive or negative acknowledgement message, in the event the *ioctl*() command sent downstream fails. For these cases, I_STR shall fail with *errno* set to the value in the message.

I_SWROPT
: Sets the write mode using the value of the argument *arg*. Valid bit settings for *arg* are:

  SNDZERO
  : Send a zero-length message downstream when a *write*() of 0 bytes occurs. To not send a zero-length message when a *write*() of 0 bytes occurs, the application shall ensure that this bit is not set in *arg* (for example, *arg* would be set to 0).

  The *ioctl*() function with the I_SWROPT command shall fail if:

  [EINVAL]
  : *arg* is not the above value.

I_GWROPT
: Returns the current write mode setting, as described above, in the **int** that is pointed to by the argument *arg*.

I_SENDFD
: Creates a new reference to the open file description associated with the file descriptor *arg*, and writes a message on the STREAMS-based pipe *fildes* containing this reference, together with the user ID and group ID of the calling process.

  The *ioctl*() function with the I_SENDFD command shall fail if:

  [EAGAIN]
  : The sending STREAM is unable to allocate a message block to contain the file pointer; or the read queue of the receiving STREAM head is full and cannot accept the message sent by I_SENDFD.

[EBADF] The *arg* argument is not a valid, open file descriptor.

[EINVAL] The *fildes* argument is not connected to a STREAM pipe.

[ENXIO] Hangup received on *fildes*.

I_RECVFD Retrieves the reference to an open file description from a message written to a STREAMS-based pipe using the I_SENDFD command, and allocates a new file descriptor in the calling process that refers to this open file description. The *arg* argument is a pointer to a **strrecvfd** data structure as defined in **<stropts.h>**.

The *fd* member is a file descriptor. The *uid* and *gid* members are the effective user ID and effective group ID, respectively, of the sending process.

If O_NONBLOCK is not set, I_RECVFD shall block until a message is present at the STREAM head. If O_NONBLOCK is set, I_RECVFD shall fail with *errno* set to [EAGAIN] if no message is present at the STREAM head.

If the message at the STREAM head is a message sent by an I_SENDFD, a new file descriptor shall be allocated for the open file descriptor referenced in the message. The new file descriptor is placed in the *fd* member of the **strrecvfd** structure pointed to by *arg*.

The *ioctl*( ) function with the I_RECVFD command shall fail if:

[EAGAIN] A message is not present at the STREAM head read queue and the O_NONBLOCK flag is set.

[EBADMSG] The message at the STREAM head read queue is not a message containing a passed file descriptor.

[EMFILE] The process has the maximum number of file descriptors currently open that it is allowed.

[ENXIO] Hangup received on *fildes*.

I_LIST Allows the process to list all the module names on the STREAM, up to and including the topmost driver name. If *arg* is a null pointer, the return value shall be the number of modules, including the driver, that are on the STREAM pointed to by *fildes*. This lets the process allocate enough space for the module names. Otherwise, it should point to a **str_list** structure.

The *sl_nmods* member indicates the number of entries the process has allocated in the array. Upon return, the *sl_modlist* member of the **str_list** structure shall contain the list of module names, and the number of entries that have been filled into the *sl_modlist* array is found in the *sl_nmods* member (the number includes the number of modules including the driver). The return value from *ioctl*( ) shall be 0. The entries are filled in starting at the top of the STREAM and continuing downstream until either the end of the STREAM is reached, or the number of requested modules (*sl_nmods*) is satisfied.

The *ioctl*( ) function with the I_LIST command shall fail if:

[EINVAL] The *sl_nmods* member is less than 1.

[EAGAIN] or [ENOSR]
Unable to allocate buffers.

I_ATMARK Allows the process to see if the message at the head of the STREAM head read queue is marked by some module downstream. The *arg* argument determines

how the checking is done when there may be multiple marked messages on the STREAM head read queue. It may take on the following values:

ANYMARK Check if the message is marked.

LASTMARK Check if the message is the last one marked on the queue.

The bitwise-inclusive OR of the flags ANYMARK and LASTMARK is permitted.

The return value shall be 1 if the mark condition is satisfied; otherwise, the value shall be 0.

The *ioctl*( ) function with the I_ATMARK command shall fail if:

[EINVAL] Invalid *arg* value.

I_CKBAND Checks if the message of a given priority band exists on the STREAM head read queue. This shall return 1 if a message of the given priority exists, 0 if no such message exists, or –1 on error. *arg* should be of type **int**.

The *ioctl*( ) function with the I_CKBAND command shall fail if:

[EINVAL] Invalid *arg* value.

I_GETBAND Returns the priority band of the first message on the STREAM head read queue in the integer referenced by *arg*.

The *ioctl*( ) function with the I_GETBAND command shall fail if:

[ENODATA] No message on the STREAM head read queue.

I_CANPUT Checks if a certain band is writable. *arg* is set to the priority band in question. The return value shall be 0 if the band is flow-controlled, 1 if the band is writable, or –1 on error.

The *ioctl*( ) function with the I_CANPUT command shall fail if:

[EINVAL] Invalid *arg* value.

I_SETCLTIME This request allows the process to set the time the STREAM head shall delay when a STREAM is closing and there is data on the write queues. Before closing each module or driver, if there is data on its write queue, the STREAM head shall delay for the specified amount of time to allow the data to drain. If, after the delay, data is still present, it shall be flushed. The *arg* argument is a pointer to an integer specifying the number of milliseconds to delay, rounded up to the nearest valid value. If I_SETCLTIME is not performed on a STREAM, an implementation-defined default timeout interval is used.

The *ioctl*( ) function with the I_SETCLTIME command shall fail if:

[EINVAL] Invalid *arg* value.

I_GETCLTIME Returns the close time delay in the integer pointed to by *arg*.

20420 **Multiplexed STREAMS Configurations**

20421 The following commands are used for connecting and disconnecting multiplexed STREAMS
20422 configurations. These commands use an implementation-defined default timeout interval.

20423 I_LINK Connects two STREAMs, where *fildes* is the file descriptor of the STREAM
20424 connected to the multiplexing driver, and *arg* is the file descriptor of the
20425 STREAM connected to another driver. The STREAM designated by *arg* is
20426 connected below the multiplexing driver. I_LINK requires the multiplexing
20427 driver to send an acknowledgement message to the STREAM head regarding
20428 the connection. This call shall return a multiplexer ID number (an identifier
20429 used to disconnect the multiplexer; see I_UNLINK) on success, and –1 on
20430 failure.

20431 The *ioctl*( ) function with the I_LINK command shall fail if:

20432 [ENXIO] Hangup received on *fildes*.

20433 [ETIME] Timeout before acknowledgement message was received at
20434 STREAM head.

20435 [EAGAIN] or [ENOSR]
20436 Unable to allocate STREAMS storage to perform the
20437 I_LINK.

20438 [EBADF] The *arg* argument is not a valid, open file descriptor.

20439 [EINVAL] The *fildes* argument does not support multiplexing; or *arg* is
20440 not a STREAM or is already connected downstream from a
20441 multiplexer; or the specified I_LINK operation would
20442 connect the STREAM head in more than one place in the
20443 multiplexed STREAM.

20444 An I_LINK can also fail while waiting for the multiplexing driver to
20445 acknowledge the request, if a message indicating an error or a hangup is
20446 received at the STREAM head of *fildes*. In addition, an error code can be
20447 returned in the positive or negative acknowledgement message. For these
20448 cases, I_LINK fails with *errno* set to the value in the message.

20449 I_UNLINK Disconnects the two STREAMs specified by *fildes* and *arg*. *fildes* is the file
20450 descriptor of the STREAM connected to the multiplexing driver. The *arg*
20451 argument is the multiplexer ID number that was returned by the I_LINK
20452 *ioctl*( ) command when a STREAM was connected downstream from the
20453 multiplexing driver. If *arg* is MUXID_ALL, then all STREAMs that were
20454 connected to *fildes* shall be disconnected. As in I_LINK, this command
20455 requires acknowledgement.

20456 The *ioctl*( ) function with the I_UNLINK command shall fail if:

20457 [ENXIO] Hangup received on *fildes*.

20458 [ETIME] Timeout before acknowledgement message was received at
20459 STREAM head.

20460 [EAGAIN] or [ENOSR]
20461 Unable to allocate buffers for the acknowledgement
20462 message.

20463 [EINVAL] Invalid multiplexer ID number.

An I_UNLINK can also fail while waiting for the multiplexing driver to acknowledge the request if a message indicating an error or a hangup is received at the STREAM head of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_UNLINK shall fail with *errno* set to the value in the message.

I_PLINK Creates a *persistent connection* between two STREAMs, where *fildes* is the file descriptor of the STREAM connected to the multiplexing driver, and *arg* is the file descriptor of the STREAM connected to another driver. This call shall create a persistent connection which can exist even if the file descriptor *fildes* associated with the upper STREAM to the multiplexing driver is closed. The STREAM designated by *arg* gets connected via a persistent connection below the multiplexing driver. I_PLINK requires the multiplexing driver to send an acknowledgement message to the STREAM head. This call shall return a multiplexer ID number (an identifier that may be used to disconnect the multiplexer; see I_PUNLINK) on success, and –1 on failure.

The *ioctl*( ) function with the I_PLINK command shall fail if:

[ENXIO] Hangup received on *fildes*.

[ETIME] Timeout before acknowledgement message was received at STREAM head.

[EAGAIN] or [ENOSR]
Unable to allocate STREAMS storage to perform the I_PLINK.

[EBADF] The *arg* argument is not a valid, open file descriptor.

[EINVAL] The *fildes* argument does not support multiplexing; or *arg* is not a STREAM or is already connected downstream from a multiplexer; or the specified I_PLINK operation would connect the STREAM head in more than one place in the multiplexed STREAM.

An I_PLINK can also fail while waiting for the multiplexing driver to acknowledge the request, if a message indicating an error or a hangup is received at the STREAM head of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_PLINK shall fail with *errno* set to the value in the message.

I_PUNLINK Disconnects the two STREAMs specified by *fildes* and *arg* from a persistent connection. The *fildes* argument is the file descriptor of the STREAM connected to the multiplexing driver. The *arg* argument is the multiplexer ID number that was returned by the I_PLINK *ioctl*( ) command when a STREAM was connected downstream from the multiplexing driver. If *arg* is MUXID_ALL, then all STREAMs which are persistent connections to *fildes* shall be disconnected. As in I_PLINK, this command requires the multiplexing driver to acknowledge the request.

The *ioctl*( ) function with the I_PUNLINK command shall fail if:

[ENXIO] Hangup received on *fildes*.

[ETIME] Timeout before acknowledgement message was received at STREAM head.

[EAGAIN] or [ENOSR]
Unable to allocate buffers for the acknowledgement message.

[EINVAL] Invalid multiplexer ID number.

An I_PUNLINK can also fail while waiting for the multiplexing driver to acknowledge the request if a message indicating an error or a hangup is received at the STREAM head of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_PUNLINK shall fail with *errno* set to the value in the message.

RETURN VALUE

Upon successful completion, *ioctl*( ) shall return a value other than –1 that depends upon the STREAMS device control function. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

Under the following general conditions, *ioctl*( ) shall fail if:

[EBADF] The *fildes* argument is not a valid open file descriptor.

[EINTR] A signal was caught during the *ioctl*( ) operation.

[EINVAL] The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

If an underlying device driver detects an error, then *ioctl*( ) shall fail if:

[EINVAL] The *request* or *arg* argument is not valid for this device.

[EIO] Some physical I/O error has occurred.

[ENOTTY] The *fildes* argument is not associated with a STREAMS device that accepts control functions.

[ENXIO] The *request* and *arg* arguments are valid for this device driver, but the service requested cannot be performed on this particular sub-device.

[ENODEV] The *fildes* argument refers to a valid STREAMS device, but the corresponding device driver does not support the *ioctl*( ) function.

If a STREAM is connected downstream from a multiplexer, any *ioctl*( ) command except I_UNLINK and I_PUNLINK shall set *errno* to [EINVAL].

EXAMPLES

None.

APPLICATION USAGE

The implementation-defined timeout interval for STREAMS has historically been 15 seconds.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *fcntl*( ), *getmsg*( ), *open*( ), *pipe*( ), *poll*( ), *putmsg*( ), *read*( ), *sigaction*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**, Section 2.6 (on page 488)

20550 **CHANGE HISTORY**

20551 First released in Issue 4, Version 2.

20552 **Issue 5**

20553 Moved from X/OPEN UNIX extension to BASE.

20554 **Issue 6**

20555 The Open Group Corrigendum U028/4 is applied, correcting text in the I_FDINSERT, [EINVAL]
20556 case to refer to *ctlbuf*.

20557 This function is marked as part of the XSI STREAMS Option Group.

20558 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

20559 **NAME**

20560 isalnum — test for an alphanumeric character

20561 **SYNOPSIS**

20562
```
#include <ctype.h>
```

20563
```
int isalnum(int c);
```

20564 **DESCRIPTION**

20565 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20566 conflict between the requirements described here and the ISO C standard is unintentional. This
20567 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20568 The *isalnum*( ) function shall test whether *c* is a character of class **alpha** or **digit** in the program's |
20569 current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

20570 The *c* argument is an **int**, the value of which the application shall ensure is representable as an |
20571 **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the
20572 behavior is undefined.

20573 **RETURN VALUE**

20574 The *isalnum*( ) function shall return non-zero if *c* is an alphanumeric character; otherwise, it shall
20575 return 0.

20576 **ERRORS**

20577 No errors are defined.

20578 **EXAMPLES**

20579 None.

20580 **APPLICATION USAGE**

20581 To ensure applications portability, especially across natural languages, only this function and
20582 those listed in the SEE ALSO section should be used for character classification.

20583 **RATIONALE**

20584 None.

20585 **FUTURE DIRECTIONS**

20586 None.

20587 **SEE ALSO**

20588 *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ),
20589 *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, **<stdio.h>**, the Base
20590 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

20591 **CHANGE HISTORY**

20592 First released in Issue 1. Derived from Issue 1 of the SVID.

20593 **Issue 6**

20594 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

20595 **NAME**

20596 isalpha — test for an alphabetic character

20597 **SYNOPSIS**

20598
```
#include <ctype.h>
```

20599
```
int isalpha(int c);
```

20600 **DESCRIPTION**

20601 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20602 conflict between the requirements described here and the ISO C standard is unintentional. This
20603 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20604 The *isalpha*( ) function shall test whether *c* is a character of class **alpha** in the program's current
20605 locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

20606 The *c* argument is an **int**, the value of which the application shall ensure is representable as an |
20607 **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the
20608 behavior is undefined.

20609 **RETURN VALUE**

20610 The *isalpha*( ) function shall return non-zero if *c* is an alphabetic character; otherwise, it shall
20611 return 0.

20612 **ERRORS**

20613 No errors are defined.

20614 **EXAMPLES**

20615 None.

20616 **APPLICATION USAGE**

20617 To ensure applications portability, especially across natural languages, only this function and
20618 those listed in the SEE ALSO section should be used for character classification.

20619 **RATIONALE**

20620 None.

20621 **FUTURE DIRECTIONS**

20622 None.

20623 **SEE ALSO**

20624 *isalnum*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
20625 *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, **<stdio.h>**,
20626 the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

20627 **CHANGE HISTORY**

20628 First released in Issue 1. Derived from Issue 1 of the SVID.

20629 **Issue 6**

20630 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

20631 **NAME**

20632 isascii — test for a 7-bit US-ASCII character

20633 **SYNOPSIS**

20634 XSI
```
#include <ctype.h>

int isascii(int c);
```
20635

20636

20637 **DESCRIPTION**

20638 The *isascii*( ) function shall test whether *c* is a 7-bit US-ASCII character code.

20639 The *isascii*( ) function is defined on all integer values.

20640 **RETURN VALUE**

20641 The *isascii*( ) function shall return non-zero if *c* is a 7-bit US-ASCII character code between 0 and
20642 octal 0177 inclusive; otherwise, it shall return 0.

20643 **ERRORS**

20644 No errors are defined.

20645 **EXAMPLES**

20646 None.

20647 **APPLICATION USAGE**

20648 None.

20649 **RATIONALE**

20650 None.

20651 **FUTURE DIRECTIONS**

20652 None.

20653 **SEE ALSO**

20654 The Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**

20655 **CHANGE HISTORY**

20656 First released in Issue 1. Derived from Issue 1 of the SVID.

20657 **NAME**

20658 isastream — test a file descriptor (**STREAMS**)

20659 **SYNOPSIS**

20660 XSR
```
#include <stropts.h>
```

20661
```
int isastream(int fildes);
```

20662

20663 **DESCRIPTION**

20664 The *isastream*( ) function shall test whether *fildes*, an open file descriptor, is associated with a
20665 STREAMS-based file.

20666 **RETURN VALUE**

20667 Upon successful completion, *isastream*( ) shall return 1 if *fildes* refers to a STREAMS-based file
20668 and 0 if not. Otherwise, *isastream*( ) shall return –1 and set *errno* to indicate the error.

20669 **ERRORS**

20670 The *isastream*( ) function shall fail if:

20671 [EBADF] The *fildes* argument is not a valid open file descriptor.

20672 **EXAMPLES**

20673 None.

20674 **APPLICATION USAGE**

20675 None.

20676 **RATIONALE**

20677 None.

20678 **FUTURE DIRECTIONS**

20679 None.

20680 **SEE ALSO**

20681 The Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**

20682 **CHANGE HISTORY**

20683 First released in Issue 4, Version 2.

20684 **Issue 5**

20685 Moved from X/OPEN UNIX extension to BASE.

20686 **NAME**

20687 isatty — test for a terminal device

20688 **SYNOPSIS**

20689
```
#include <unistd.h>
```

20690
```
int isatty(int fildes);
```

20691 **DESCRIPTION**

20692 The *isatty*( ) function shall test whether *fildes*, an open file descriptor, is associated with a
20693 terminal device.

20694 **RETURN VALUE**

20695 The *isatty*( ) function shall return 1 if *fildes* is associated with a terminal; otherwise, it shall return
20696 0 and may set *errno* to indicate the error.

20697 **ERRORS**

20698 The *isatty*( ) function may fail if:

20699 [EBADF] The *fildes* argument is not a valid open file descriptor.

20700 [ENOTTY] The *fildes* argument is not associated with a terminal.

20701 **EXAMPLES**

20702 None.

20703 **APPLICATION USAGE**

20704 The *isatty*( ) function does not necessarily indicate that a human being is available for interaction
20705 via *fildes*. It is quite possible that non-terminal devices are connected to the communications
20706 line.

20707 **RATIONALE**

20708 None.

20709 **FUTURE DIRECTIONS**

20710 None.

20711 **SEE ALSO**

20712 The Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

20713 **CHANGE HISTORY**

20714 First released in Issue 1. Derived from Issue 1 of the SVID.

20715 **Issue 6**

20716 The following new requirements on POSIX implementations derive from alignment with the
20717 Single UNIX Specification:

20718 - The optional setting of *errno* to indicate an error is added.

20719 - The [EBADF] and [ENOTTY] optional error conditions are added.

20720 **NAME**

20721 isblank — test for a blank character

20722 **SYNOPSIS**

20723
```
#include <ctype.h>
```

20724
```
int isblank(int c);
```

20725 **DESCRIPTION**

20726 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20727 conflict between the requirements described here and the ISO C standard is unintentional. This
20728 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20729 The *isblank*( ) function shall test whether *c* is a character of class **blank** in the program's current
20730 locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

20731 The *c* argument is a type **int**, the value of which the application shall ensure is a character |
20732 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
20733 any other value, the behavior is undefined.

20734 **RETURN VALUE**

20735 The *isblank*( ) function shall return non-zero if *c* is a <blank>; otherwise, it shall return 0.

20736 **ERRORS**

20737 No errors are defined.

20738 **EXAMPLES**

20739 None.

20740 **APPLICATION USAGE**

20741 To ensure applications portability, especially across natural languages, only this function and
20742 those listed in the SEE ALSO section should be used for character classification.

20743 **RATIONALE**

20744 None.

20745 **FUTURE DIRECTIONS**

20746 None.

20747 **SEE ALSO**

20748 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
20749 *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**

20750 **CHANGE HISTORY**

20751 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

20752 **NAME**

20753 iscntrl — test for a control character

20754 **SYNOPSIS**

20755
```
#include <ctype.h>
```

20756
```
int iscntrl(int c);
```

20757 **DESCRIPTION**

20758 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20759 conflict between the requirements described here and the ISO C standard is unintentional. This
20760 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20761 The *iscntrl*( ) function shall test whether *c* is a character of class **cntrl** in the program's current
20762 locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

20763 The *c* argument is a type **int**, the value of which the application shall ensure is a character |
20764 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
20765 any other value, the behavior is undefined.

20766 **RETURN VALUE**

20767 The *iscntrl*( ) function shall return non-zero if *c* is a control character; otherwise, it shall return 0.

20768 **ERRORS**

20769 No errors are defined.

20770 **EXAMPLES**

20771 None.

20772 **APPLICATION USAGE**

20773 To ensure applications portability, especially across natural languages, only this function and
20774 those listed in the SEE ALSO section should be used for character classification.

20775 **RATIONALE**

20776 None.

20777 **FUTURE DIRECTIONS**

20778 None.

20779 **SEE ALSO**

20780 *isalnum*( ), *isalpha*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
20781 *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base
20782 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

20783 **CHANGE HISTORY**

20784 First released in Issue 1. Derived from Issue 1 of the SVID.

20785 **Issue 6**

20786 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

20787 **NAME**

20788 isdigit — test for a decimal digit

20789 **SYNOPSIS**

20790
```
#include <ctype.h>
```

20791
```
int isdigit(int c);
```

20792 **DESCRIPTION**

20793 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20794 conflict between the requirements described here and the ISO C standard is unintentional. This
20795 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20796 The *isdigit*( ) function shall test whether *c* is a character of class **digit** in the program's current
20797 locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

20798 The *c* argument is an **int**, the value of which the application shall ensure is a character |
20799 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
20800 any other value, the behavior is undefined.

20801 **RETURN VALUE**

20802 The *isdigit*( ) function shall return non-zero if *c* is a decimal digit; otherwise, it shall return 0.

20803 **ERRORS**

20804 No errors are defined.

20805 **EXAMPLES**

20806 None.

20807 **APPLICATION USAGE**

20808 To ensure applications portability, especially across natural languages, only this function and
20809 those listed in the SEE ALSO section should be used for character classification.

20810 **RATIONALE**

20811 None.

20812 **FUTURE DIRECTIONS**

20813 None.

20814 **SEE ALSO**

20815 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
20816 *isxdigit*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**

20817 **CHANGE HISTORY**

20818 First released in Issue 1. Derived from Issue 1 of the SVID.

20819 **Issue 6**

20820 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

20821 **NAME**

20822 isfinite — test for finite value

20823 **SYNOPSIS**

20824 `#include <math.h>`

20825 `int isfinite(real-floating x);`

20826 **DESCRIPTION**

20827 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20828 conflict between the requirements described here and the ISO C standard is unintentional. This
20829 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20830 The *isfinite*( ) macro shall determine whether its argument has a finite value (zero, subnormal, or
20831 normal, and not infinite or NaN). First, an argument represented in a format wider than its
20832 semantic type is converted to its semantic type. Then determination is based on the type of the
20833 argument.

20834 **RETURN VALUE**

20835 The *isfinite*( ) macro shall return a non-zero value if and only if its argument has a finite value.

20836 **ERRORS**

20837 No errors are defined.

20838 **EXAMPLES**

20839 None.

20840 **APPLICATION USAGE**

20841 None.

20842 **RATIONALE**

20843 None.

20844 **FUTURE DIRECTIONS**

20845 None.

20846 **SEE ALSO**

20847 *fpclassify*( ), *isinf*( ), *isnan*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of
20848 IEEE Std 1003.1-200x **<math.h>**

20849 **CHANGE HISTORY**

20850 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

20851 **NAME**

20852 isgraph — test for a visible character

20853 **SYNOPSIS**

20854
```
#include <ctype.h>
```

20855
```
int isgraph(int c);
```

20856 **DESCRIPTION**

20857 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20858 conflict between the requirements described here and the ISO C standard is unintentional. This
20859 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20860 The *isgraph*( ) function shall test whether *c* is a character of class **graph** in the program's current
20861 locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

20862 The *c* argument is an **int**, the value of which the application shall ensure is a character |
20863 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
20864 any other value, the behavior is undefined.

20865 **RETURN VALUE**

20866 The *isgraph*( ) function shall return non-zero if *c* is a character with a visible representation;
20867 otherwise, it shall return 0.

20868 **ERRORS**

20869 No errors are defined.

20870 **EXAMPLES**

20871 None.

20872 **APPLICATION USAGE**

20873 To ensure applications portability, especially across natural languages, only this function and
20874 those listed in the SEE ALSO section should be used for character classification.

20875 **RATIONALE**

20876 None.

20877 **FUTURE DIRECTIONS**

20878 None.

20879 **SEE ALSO**

20880 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ),
20881 *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions
20882 volume of IEEE Std 1003.1-200x, Chapter 7, Locale

20883 **CHANGE HISTORY**

20884 First released in Issue 1. Derived from Issue 1 of the SVID.

20885 **Issue 6**

20886 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

20887 **NAME**

20888 isgreater — test if x greater than y

20889 **SYNOPSIS**

20890
```
#include <math.h>
```

20891
```
int isgreater(real-floating x, real-floating y);
```

20892 **DESCRIPTION**

20893 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20894 conflict between the requirements described here and the ISO C standard is unintentional. This
20895 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20896 The *isgreater*( ) macro shall determine whether its first argument is greater than its second
20897 argument. The value of *isgreater*(*x*, *y*) shall be equal to (*x*) > (*y*); however, unlike (*x*) > (*y*),
20898 *isgreater*(*x*, *y*) shall not raise the invalid floating-point exception when *x* and *y* are unordered.

20899 **RETURN VALUE**

20900 Upon successful completion, the *isgreater*( ) macro shall return the value of (*x*) > (*y*).

20901 If *x* or *y* is NaN, 0 shall be returned.

20902 **ERRORS**

20903 No errors are defined.

20904 **EXAMPLES**

20905 None.

20906 **APPLICATION USAGE**

20907 The relational and equality operators support the usual mathematical relationships between
20908 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
20909 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
20910 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
20911 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
20912 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
20913 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
20914 indicates that the argument shall be an expression of **real-floating** type.

20915 **RATIONALE**

20916 None.

20917 **FUTURE DIRECTIONS**

20918 None.

20919 **SEE ALSO**

20920 *isgreaterequal*( ), *isless*( ), *islessequal*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume of
20921 IEEE Std 1003.1-200x **<math.h>**

20922 **CHANGE HISTORY**

20923 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

20924 **NAME**

20925 isgreaterequal — test if x greater than or equal to y

20926 **SYNOPSIS**

20927 `#include <math.h>`

20928 `int isgreaterequal(real-floating x, real-floating y);`

20929 **DESCRIPTION**

20930 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20931 conflict between the requirements described here and the ISO C standard is unintentional. This
20932 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20933 The *isgreaterequal*( ) macro shall determine whether its first argument is greater than or equal to
20934 its second argument. The value of *isgreaterequal*(*x*, *y*) shall be equal to (*x*) >= (*y*); however, unlike
20935 (*x*) >= (*y*), *isgreaterequal*(*x*, *y*) shall not raise the invalid floating-point exception when *x* and *y* are
20936 unordered.

20937 **RETURN VALUE**

20938 Upon successful completion, the *isgreaterequal*( ) macro shall return the value of (*x*) >= (*y*).

20939 If *x* or *y* is NaN, 0 shall be returned.

20940 **ERRORS**

20941 No errors are defined.

20942 **EXAMPLES**

20943 None.

20944 **APPLICATION USAGE**

20945 The relational and equality operators support the usual mathematical relationships between
20946 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
20947 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
20948 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
20949 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
20950 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
20951 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
20952 indicates that the argument shall be an expression of **real-floating** type.

20953 **RATIONALE**

20954 None.

20955 **FUTURE DIRECTIONS**

20956 None.

20957 **SEE ALSO**

20958 *isgreater*( ), *isless*( ), *islessequal*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume of
20959 IEEE Std 1003.1-200x **<math.h>**

20960 **CHANGE HISTORY**

20961 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

20962 **NAME**

20963 isinf — test for infinity

20964 **SYNOPSIS**

20965 `#include <math.h>`

20966 `int isinf(real-floating x);`

20967 **DESCRIPTION**

20968 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20969 conflict between the requirements described here and the ISO C standard is unintentional. This
20970 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20971 The *isinf*( ) macro shall determine whether its argument value is an infinity (positive or
20972 negative). First, an argument represented in a format wider than its semantic type is converted
20973 to its semantic type. Then determination is based on the type of the argument.

20974 **RETURN VALUE**

20975 The *isinf*( ) macro shall return a non-zero value if and only if its argument has an infinite value.

20976 **ERRORS**

20977 No errors are defined.

20978 **EXAMPLES**

20979 None.

20980 **APPLICATION USAGE**

20981 None.

20982 **RATIONALE**

20983 None.

20984 **FUTURE DIRECTIONS**

20985 None.

20986 **SEE ALSO**

20987 *fpclassify*( ), *isfinite*( ), *isnan*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of
20988 IEEE Std 1003.1-200x **<math.h>**

20989 **CHANGE HISTORY**

20990 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

20991 **NAME**

20992 isless — test if x is less than y

20993 **SYNOPSIS**

20994 `#include <math.h>`

20995 `int isless(real-floating x, real-floating y);`

20996 **DESCRIPTION**

20997 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20998 conflict between the requirements described here and the ISO C standard is unintentional. This
20999 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21000 The *isless*( ) macro shall determine whether its first argument is less than its second argument.
21001 The value of *isless*(*x*, *y*) shall be equal to (*x*) < (*y*); however, unlike (*x*) < (*y*), *isless*(*x*, *y*) shall not
21002 raise the invalid floating-point exception when *x* and *y* are unordered.

21003 **RETURN VALUE**

21004 Upon successful completion, the *isless*( ) macro shall return the value of (*x*) < (*y*).

21005 If *x* or *y* is NaN, 0 shall be returned.

21006 **ERRORS**

21007 No errors are defined.

21008 **EXAMPLES**

21009 None.

21010 **APPLICATION USAGE**

21011 The relational and equality operators support the usual mathematical relationships between
21012 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21013 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21014 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21015 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21016 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21017 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21018 indicates that the argument shall be an expression of **real-floating** type.

21019 **RATIONALE**

21020 None.

21021 **FUTURE DIRECTIONS**

21022 None.

21023 **SEE ALSO**

21024 *isgreater*( ), *isgreaterequal*( ), *islessequal*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume
21025 of IEEE Std 1003.1-200x, **<math.h>**

21026 **CHANGE HISTORY**

21027 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

21028 **NAME**

21029 islessequal — test if x is less than or equal to y

21030 **SYNOPSIS**

21031 `#include <math.h>`

21032 `int islessequal(real-floating x, real-floating y);`

21033 **DESCRIPTION**

21034 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21035 conflict between the requirements described here and the ISO C standard is unintentional. This
21036 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21037 The *islessequal*( ) macro shall determine whether its first argument is less than or equal to its
21038 second argument. The value of *islessequal*(*x*, *y*) shall be equal to (*x*) <= (*y*); however, unlike
21039 (*x*) <= (*y*), *islessequal*(*x*, *y*) shall not raise the invalid floating-point exception when *x* and *y* are
21040 unordered.

21041 **RETURN VALUE**

21042 Upon successful completion, the *islessequal*( ) macro shall return the value of (*x*) <= (*y*).

21043 If *x* or *y* is NaN, 0 shall be returned.

21044 **ERRORS**

21045 No errors are defined.

21046 **EXAMPLES**

21047 None.

21048 **APPLICATION USAGE**

21049 The relational and equality operators support the usual mathematical relationships between
21050 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21051 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21052 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21053 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21054 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21055 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21056 indicates that the argument shall be an expression of **real-floating** type.

21057 **RATIONALE**

21058 None.

21059 **FUTURE DIRECTIONS**

21060 None.

21061 **SEE ALSO**

21062 *isgreater*( ), *isgreaterequal*( ), *isless*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume of
21063 IEEE Std 1003.1-200x **<math.h>**

21064 **CHANGE HISTORY**

21065 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

21066 **NAME**

21067 islessgreater — test if x is less than or greater than y

21068 **SYNOPSIS**

21069
```
#include <math.h>
```

21070
```
int islessgreater(real-floating x, real-floating y);
```

21071 **DESCRIPTION**

21072 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21073 conflict between the requirements described here and the ISO C standard is unintentional. This
21074 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21075 The *islessgreater*( ) macro shall determine whether its first argument is less than or greater than
21076 its second argument. The *islessgreater*(*x*, *y*) macro is similar to (*x*) < (*y*) | | (*x*) > (*y*); however,
21077 *islessgreater*(*x*, *y*) shall not raise the invalid floating-point exception when *x* and *y* are unordered
21078 (nor shall it evaluate *x* and *y* twice).

21079 **RETURN VALUE**

21080 Upon successful completion, the *islessgreater*( ) macro shall return the value of
21081 (*x*) < (*y*) | | (*x*) > (*y*).

21082 If *x* or *y* is NaN, 0 shall be returned.

21083 **ERRORS**

21084 No errors are defined.

21085 **EXAMPLES**

21086 None.

21087 **APPLICATION USAGE**

21088 The relational and equality operators support the usual mathematical relationships between
21089 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21090 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21091 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21092 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21093 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21094 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21095 indicates that the argument shall be an expression of **real-floating** type.

21096 **RATIONALE**

21097 None.

21098 **FUTURE DIRECTIONS**

21099 None.

21100 **SEE ALSO**

21101 *isgreater*( ), *isgreaterequal*( ), *isless*( ), *islessequal*( ), *isunordered*( ), the Base Definitions volume of
21102 IEEE Std 1003.1-200x **<math.h>**

21103 **CHANGE HISTORY**

21104 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

21105 **NAME**

21106 islower — test for a lowercase letter

21107 **SYNOPSIS**

21108
```
#include <ctype.h>
```

21109
```
int islower(int c);
```

21110 **DESCRIPTION**

21111 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21112 conflict between the requirements described here and the ISO C standard is unintentional. This
21113 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21114 The *islower*( ) function shall test whether *c* is a character of class **lower** in the program's current
21115 locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21116 The *c* argument is an **int**, the value of which the application shall ensure is a character |
21117 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21118 any other value, the behavior is undefined.

21119 **RETURN VALUE**

21120 The *islower*( ) function shall return non-zero if *c* is a lowercase letter; otherwise, it shall return 0.

21121 **ERRORS**

21122 No errors are defined.

21123 **EXAMPLES**

21124 **Testing for a Lowercase Letter**

21125 The following example tests whether the value is a lowercase letter, based on the locale of the
21126 user, then uses it as part of a key value.

```
21127 #include <ctype.h>
21128 #include <stdlib.h>
21129 #include <locale.h>
21130 ...
21131 char *keystr;
21132 int elementlen, len;
21133 char c;
21134 ...
21135 setlocale(LC_ALL, "");
21136 ...
21137 len = 0;
21138 while (len < elementlen) {
21139     c = (char) (rand() % 256);
21140 ...
21141     if (islower(c))
21142         keystr[len++] = c;
21143     }
21144 ...
```

21145 **APPLICATION USAGE**

21146 To ensure applications portability, especially across natural languages, only this function and
21147 those listed in the SEE ALSO section should be used for character classification.

21148 **RATIONALE**

21149 None.

21150 **FUTURE DIRECTIONS**

21151 None.

21152 **SEE ALSO**

21153 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
21154 *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base
21155 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

21156 **CHANGE HISTORY**

21157 First released in Issue 1. Derived from Issue 1 of the SVID.

21158 **Issue 6**

21159 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

21160 An example is added. |

21161 **NAME**

21162 isnan — test for a NaN

21163 **SYNOPSIS**

21164
```
#include <math.h>
```

21165
```
int isnan(real-floating x);
```

21166 **DESCRIPTION**

21167 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21168 conflict between the requirements described here and the ISO C standard is unintentional. This
21169 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21170 The *isnan*( ) macro shall determine whether its argument value is a NaN. First, an argument
21171 represented in a format wider than its semantic type is converted to its semantic type. Then
21172 determination is based on the type of the argument.

21173 **RETURN VALUE**

21174 The *isnan*( ) macro shall return a non-zero value if and only if its argument has a NaN value.

21175 **ERRORS**

21176 No errors are defined.

21177 **EXAMPLES**

21178 None.

21179 **APPLICATION USAGE**

21180 None.

21181 **RATIONALE**

21182 None.

21183 **FUTURE DIRECTIONS**

21184 None.

21185 **SEE ALSO**

21186 *fpclassify*( ), *isfinite*( ), *isinf*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of
21187 IEEE Std 1003.1-200x, **<math.h>**

21188 **CHANGE HISTORY**

21189 First released in Issue 3.

21190 **Issue 5**

21191 The DESCRIPTION is updated to indicate the return value when NaN is not supported. This
21192 text was previously published in the APPLICATION USAGE section.

21193 **Issue 6**

21194 Entry re-written for alignment with the ISO / IEC 9899: 1999 standard.

21195 **NAME**

21196 isnormal — test for a normal value

21197 **SYNOPSIS**

21198
```
#include <math.h>
```

21199
```
int isnormal(real-floating x);
```

21200 **DESCRIPTION**

21201 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21202 conflict between the requirements described here and the ISO C standard is unintentional. This
21203 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21204 The *isnormal*() macro shall determine whether its argument value is normal (neither zero,
21205 subnormal, infinite, nor NaN). First, an argument represented in a format wider than its
21206 semantic type is converted to its semantic type. Then determination is based on the type of the
21207 argument.

21208 **RETURN VALUE**

21209 The *isnormal*() macro shall return a non-zero value if and only if its argument has a normal
21210 value.

21211 **ERRORS**

21212 No errors are defined.

21213 **EXAMPLES**

21214 None.

21215 **APPLICATION USAGE**

21216 None.

21217 **RATIONALE**

21218 None.

21219 **FUTURE DIRECTIONS**

21220 None.

21221 **SEE ALSO**

21222 *fpclassify*(), *isfinite*(), *isinf*(), *isnan*(), *signbit*(), the Base Definitions volume of
21223 IEEE Std 1003.1-200x, **<math.h>**

21224 **CHANGE HISTORY**

21225 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21226 **NAME**

21227 isprint — test for a printable character

21228 **SYNOPSIS**

21229
```
#include <ctype.h>
```

21230
```
int isprint(int c);
```

21231 **DESCRIPTION**

21232 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21233 conflict between the requirements described here and the ISO C standard is unintentional. This
21234 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21235 The *isprint*( ) function shall test whether *c* is a character of class **print** in the program's current
21236 locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21237 The *c* argument is an **int**, the value of which the application shall ensure is a character
21238 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21239 any other value, the behavior is undefined.

21240 **RETURN VALUE**

21241 The *isprint*( ) function shall return non-zero if *c* is a printable character; otherwise, it shall return
21242 0.

21243 **ERRORS**

21244 No errors are defined.

21245 **EXAMPLES**

21246 None.

21247 **APPLICATION USAGE**

21248 To ensure applications portability, especially across natural languages, only this function and
21249 those listed in the SEE ALSO section should be used for character classification.

21250 **RATIONALE**

21251 None.

21252 **FUTURE DIRECTIONS**

21253 None.

21254 **SEE ALSO**

21255 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
21256 *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base
21257 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

21258 **CHANGE HISTORY**

21259 First released in Issue 1. Derived from Issue 1 of the SVID.

21260 **Issue 6**

21261 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

21262 **NAME**

21263 ispunct — test for a punctuation character

21264 **SYNOPSIS**

21265
```
#include <ctype.h>
```

21266
```
int ispunct(int c);
```

21267 **DESCRIPTION**

21268 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21269 conflict between the requirements described here and the ISO C standard is unintentional. This
21270 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21271 The *ispunct*( ) function shall test whether *c* is a character of class **punct** in the program's current
21272 locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21273 The *c* argument is an **int**, the value of which the application shall ensure is a character |
21274 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21275 any other value, the behavior is undefined.

21276 **RETURN VALUE**

21277 The *ispunct*( ) function shall return non-zero if *c* is a punctuation character; otherwise, it shall
21278 return 0.

21279 **ERRORS**

21280 No errors are defined.

21281 **EXAMPLES**

21282 None.

21283 **APPLICATION USAGE**

21284 To ensure applications portability, especially across natural languages, only this function and
21285 those listed in the SEE ALSO section should be used for character classification.

21286 **RATIONALE**

21287 None.

21288 **FUTURE DIRECTIONS**

21289 None.

21290 **SEE ALSO**

21291 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *isspace*( ), *isupper*( ), *isxdigit*( ),
21292 *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions
21293 volume of IEEE Std 1003.1-200x, Chapter 7, Locale

21294 **CHANGE HISTORY**

21295 First released in Issue 1. Derived from Issue 1 of the SVID.

21296 **Issue 6**

21297 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

21298 **NAME**

21299 isspace — test for a white-space character

21300 **SYNOPSIS**

21301
```
#include <ctype.h>
```

21302
```
int isspace(int c);
```

21303 **DESCRIPTION**

21304 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21305 conflict between the requirements described here and the ISO C standard is unintentional. This
21306 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21307 The *isspace*( ) function shall test whether *c* is a character of class **space** in the program's current
21308 locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21309 The *c* argument is an **int**, the value of which the application shall ensure is a character |
21310 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21311 any other value, the behavior is undefined.

21312 **RETURN VALUE**

21313 The *isspace*( ) function shall return non-zero if *c* is a white-space character; otherwise, it shall
21314 return 0.

21315 **ERRORS**

21316 No errors are defined.

21317 **EXAMPLES**

21318 None.

21319 **APPLICATION USAGE**

21320 To ensure applications portability, especially across natural languages, only this function and
21321 those listed in the SEE ALSO section should be used for character classification.

21322 **RATIONALE**

21323 None.

21324 **FUTURE DIRECTIONS**

21325 None.

21326 **SEE ALSO**

21327 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isupper*( ),
21328 *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base
21329 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

21330 **CHANGE HISTORY**

21331 First released in Issue 1. Derived from Issue 1 of the SVID.

21332 **Issue 6**

21333 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

21334 **NAME**

21335 isunordered — test if arguments are unordered

21336 **SYNOPSIS**

21337
```
#include <math.h>
```

21338
```
int isunordered(real-floating x, real-floating y);
```

21339 **DESCRIPTION**

21340 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21341 conflict between the requirements described here and the ISO C standard is unintentional. This
21342 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21343 The *isunordered*( ) macro shall determine whether its arguments are unordered.

21344 **RETURN VALUE**

21345 Upon successful completion, the *isunordered*( ) macro shall return 1 if its arguments are
21346 unordered, and 0 otherwise.

21347 If *x* or *y* is NaN, 0 shall be returned.

21348 **ERRORS**

21349 No errors are defined.

21350 **EXAMPLES**

21351 None.

21352 **APPLICATION USAGE**

21353 The relational and equality operators support the usual mathematical relationships between
21354 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21355 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21356 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21357 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21358 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21359 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21360 indicates that the argument shall be an expression of **real-floating** type.

21361 **RATIONALE**

21362 None.

21363 **FUTURE DIRECTIONS**

21364 None.

21365 **SEE ALSO**

21366 *isgreater*( ), *isgreaterequal*( ), *isless*( ), *islessequal*( ), *islessgreater*( ), the Base Definitions volume of
21367 IEEE Std 1003.1-200x, **<math.h>**

21368 **CHANGE HISTORY**

21369 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

21370 **NAME**

21371 isupper — test for an uppercase letter

21372 **SYNOPSIS**

21373
```
#include <ctype.h>
```

21374
```
int isupper(int c);
```

21375 **DESCRIPTION**

21376 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21377 conflict between the requirements described here and the ISO C standard is unintentional. This
21378 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21379 The *isupper*( ) function shall test whether *c* is a character of class **upper** in the program's current
21380 locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21381 The *c* argument is an **int**, the value of which the application shall ensure is a character |
21382 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21383 any other value, the behavior is undefined.

21384 **RETURN VALUE**

21385 The *isupper*( ) function shall return non-zero if *c* is an uppercase letter; otherwise, it shall return 0.

21386 **ERRORS**

21387 No errors are defined.

21388 **EXAMPLES**

21389 None.

21390 **APPLICATION USAGE**

21391 To ensure applications portability, especially across natural languages, only this function and
21392 those listed in the SEE ALSO section should be used for character classification.

21393 **RATIONALE**

21394 None.

21395 **FUTURE DIRECTIONS**

21396 None.

21397 **SEE ALSO**

21398 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isxdigit*( ),
21399 *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions
21400 volume of IEEE Std 1003.1-200x, Chapter 7, Locale

21401 **CHANGE HISTORY**

21402 First released in Issue 1. Derived from Issue 1 of the SVID.

21403 **Issue 6**

21404 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

21405 **NAME**

21406 iswalnum — test for an alphanumeric wide-character code

21407 **SYNOPSIS**

21408 `#include <wctype.h>`

21409 `int iswalnum(wint_t wc);`

21410 **DESCRIPTION**

21411 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21412 conflict between the requirements described here and the ISO C standard is unintentional. This
21413 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21414 The *iswalnum*( ) function shall test whether *wc* is a wide-character code representing a character
21415 of class **alpha** or **digit** in the program's current locale; see the Base Definitions volume of
21416 IEEE Std 1003.1-200x, Chapter 7, Locale.

21417 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character |
21418 code corresponding to a valid character in the current locale, or equal to the value of the macro
21419 WEOF. If the argument has any other value, the behavior is undefined.

21420 **RETURN VALUE**

21421 The *iswalnum*( ) function shall return non-zero if *wc* is an alphanumeric wide-character code;
21422 otherwise, it shall return 0.

21423 **ERRORS**

21424 No errors are defined.

21425 **EXAMPLES**

21426 None.

21427 **APPLICATION USAGE**

21428 To ensure applications portability, especially across natural languages, only this function and
21429 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21430 **RATIONALE**

21431 None.

21432 **FUTURE DIRECTIONS**

21433 None.

21434 **SEE ALSO**

21435 *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
21436 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
21437 IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, **<stdio.h>**, the Base Definitions volume of
21438 IEEE Std 1003.1-200x, Chapter 7, Locale

21439 **CHANGE HISTORY**

21440 First released as a World-wide Portability Interface in Issue 4.

21441 **Issue 5**

21442 The following change has been made in this issue for alignment with
21443 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

21444 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21445 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

21448 **NAME**

21449 iswalpha — test for an alphabetic wide-character code

21450 **SYNOPSIS**

21451 `#include <wctype.h>`

21452 `int iswalpha(wint_t wc);`

21453 **DESCRIPTION**

21454 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21455 conflict between the requirements described here and the ISO C standard is unintentional. This
21456 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21457 The *iswalpha*( ) function shall test whether *wc* is a wide-character code representing a character of
21458 class **alpha** in the program's current locale; see the Base Definitions volume of
21459 IEEE Std 1003.1-200x, Chapter 7, Locale.

21460 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character |
21461 code corresponding to a valid character in the current locale, or equal to the value of the macro
21462 WEOF. If the argument has any other value, the behavior is undefined.

21463 **RETURN VALUE**

21464 The *iswalpha*( ) function shall return non-zero if *wc* is an alphabetic wide-character code;
21465 otherwise, it shall return 0.

21466 **ERRORS**

21467 No errors are defined.

21468 **EXAMPLES**

21469 None.

21470 **APPLICATION USAGE**

21471 To ensure applications portability, especially across natural languages, only this function and
21472 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21473 **RATIONALE**

21474 None.

21475 **FUTURE DIRECTIONS**

21476 None.

21477 **SEE ALSO**

21478 *iswalnum*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
21479 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
21480 IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, **<stdio.h>**, the Base Definitions volume of
21481 IEEE Std 1003.1-200x, Chapter 7, Locale

21482 **CHANGE HISTORY**

21483 First released in Issue 4.

21484 **Issue 5**

21485 The following change has been made in this issue for alignment with
21486 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

21487 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21488 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

21489 **Issue 6**

21490 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

iswblank — test for a blank wide-character code

SYNOPSIS

```
#include <wctype.h>

int iswblank(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswblank*( ) function shall test whether *wc* is a wide-character code representing a character of class **blank** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

RETURN VALUE

The *iswblank*( ) function shall return non-zero if *wc* is a blank wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**, **<wctype.h>**, **<stdio.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

21526 **NAME**

21527 iswcntrl — test for a control wide-character code

21528 **SYNOPSIS**

21529 `#include <wctype.h>`

21530 `int iswcntrl(wint_t wc);`

21531 **DESCRIPTION**

21532 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21533 conflict between the requirements described here and the ISO C standard is unintentional. This
21534 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21535 The *iswcntrl*( ) function shall test whether *wc* is a wide-character code representing a character of
21536 class **cntrl** in the program's current locale; see the Base Definitions volume of
21537 IEEE Std 1003.1-200x, Chapter 7, Locale.

21538 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
21539 code corresponding to a valid character in the current locale, or equal to the value of the macro
21540 WEOF. If the argument has any other value, the behavior is undefined.

21541 **RETURN VALUE**

21542 The *iswcntrl*( ) function shall return non-zero if *wc* is a control wide-character code; otherwise, it
21543 shall return 0.

21544 **ERRORS**

21545 No errors are defined.

21546 **EXAMPLES**

21547 None.

21548 **APPLICATION USAGE**

21549 To ensure applications portability, especially across natural languages, only this function and
21550 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21551 **RATIONALE**

21552 None.

21553 **FUTURE DIRECTIONS**

21554 None.

21555 **SEE ALSO**

21556 *iswalnum*( ), *iswalpha*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
21557 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
21558 IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of
21559 IEEE Std 1003.1-200x, Chapter 7, Locale

21560 **CHANGE HISTORY**

21561 First released in Issue 4.

21562 **Issue 5**

21563 The following change has been made in this issue for alignment with
21564 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

21565 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21566 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

21567 **Issue 6**

21568 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

21569 **NAME**

21570 iswctype — test character for a specified class

21571 **SYNOPSIS**

21572
```
#include <wctype.h>
```

21573
```
int iswctype(wint_t wc, wctype_t charclass);
```

21574 **DESCRIPTION**

21575 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21576 conflict between the requirements described here and the ISO C standard is unintentional. This
21577 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21578 The *iswctype*( ) function shall determine whether the wide-character code *wc* has the character
21579 class *charclass*, returning true or false. The *iswctype*( ) function is defined on WEOF and wide-
21580 character codes corresponding to the valid character encodings in the current locale. If the *wc*
21581 argument is not in the domain of the function, the result is undefined. If the value of *charclass* is
21582 invalid (that is, not obtained by a call to *wctype*( ) or *charclass* is invalidated by a subsequent call
21583 to *setlocale*( ) that has affected category *LC_CTYPE*) the result is unspecified.

21584 **RETURN VALUE**

21585 The *iswctype*( ) function shall return non-zero (true) if and only if *wc* has the property described
21586 CX by *charclass*. If *charclass* is 0, *iswctype*( ) shall return 0.

21587 **ERRORS**

21588 No errors are defined.

21589 **EXAMPLES**

21590 **Testing for a Valid Character**

21591
```
#include <wctype.h>
```
21592
```
...
```
21593
```
int yes_or_no;
```
21594
```
wint_t wc;
```
21595
```
wctype_t valid_class;
```
21596
```
...
```
21597
```
if ((valid_class=wctype("vowel")) == (wctype_t)0)
```
21598
```
    /* Invalid character class. */
```
21599
```
yes_or_no=iswctype(wc,valid_class);
```

21600 **APPLICATION USAGE**

21601 The twelve strings "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower",
21602 "print", "punct", "space", "upper", and "xdigit" are reserved for the standard
21603 character classes. In the table below, the functions in the left column are equivalent to the
21604 functions in the right column.

21605
```
iswalnum(wc)    iswctype(wc, wctype("alnum"))
```
21606
```
iswalpha(wc)    iswctype(wc, wctype("alpha"))
```
21607
```
iswblank(wc)    iswctype(wc, wctype("blank"))
```
21608
```
iswcntrl(wc)    iswctype(wc, wctype("cntrl"))
```
21609
```
iswdigit(wc)    iswctype(wc, wctype("digit"))
```
21610
```
iswgraph(wc)    iswctype(wc, wctype("graph"))
```
21611
```
iswlower(wc)    iswctype(wc, wctype("lower"))
```
21612
```
iswprint(wc)    iswctype(wc, wctype("print"))
```
21613
```
iswpunct(wc)    iswctype(wc, wctype("punct"))
```
21614
```
iswspace(wc)    iswctype(wc, wctype("space"))
```

```
iswupper(wc)    iswctype(wc, wctype("upper"))
iswxdigit(wc)   iswctype(wc, wctype("xdigit"))
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), *wctype*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**

CHANGE HISTORY

First released as World-wide Portability Interfaces in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The behavior of *n*=0 is now described.

An example is added.

A new function, *iswblank*( ), is added to the list in the APPLICATION USAGE.

21636 **NAME**

21637 iswdigit — test for a decimal digit wide-character code

21638 **SYNOPSIS**

21639
```
#include <wctype.h>
```

21640
```
int iswdigit(wint_t wc);
```

21641 **DESCRIPTION**

21642 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21643 conflict between the requirements described here and the ISO C standard is unintentional. This
21644 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21645 The *iswdigit*( ) function shall test whether *wc* is a wide-character code representing a character of
21646 class **digit** in the program's current locale; see the Base Definitions volume of
21647 IEEE Std 1003.1-200x, Chapter 7, Locale.

21648 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character |
21649 code corresponding to a valid character in the current locale, or equal to the value of the macro
21650 WEOF. If the argument has any other value, the behavior is undefined.

21651 **RETURN VALUE**

21652 The *iswdigit*( ) function shall return non-zero if *wc* is a decimal digit wide-character code;
21653 otherwise, it shall return 0.

21654 **ERRORS**

21655 No errors are defined.

21656 **EXAMPLES**

21657 None.

21658 **APPLICATION USAGE**

21659 To ensure applications portability, especially across natural languages, only this function and
21660 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21661 **RATIONALE**

21662 None.

21663 **FUTURE DIRECTIONS**

21664 None.

21665 **SEE ALSO**

21666 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
21667 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
21668 IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**

21669 **CHANGE HISTORY**

21670 First released in Issue 4.

21671 **Issue 5**

21672 The following change has been made in this issue for alignment with
21673 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

21674 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21675 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

21676 **Issue 6**

21677 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

21678 **NAME**

21679 iswgraph — test for a visible wide-character code

21680 **SYNOPSIS**

21681 `#include <wctype.h>`

21682 `int iswgraph(wint_t wc);`

21683 **DESCRIPTION**

21684 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21685 conflict between the requirements described here and the ISO C standard is unintentional. This
21686 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21687 The *iswgraph*( ) function shall test whether *wc* is a wide-character code representing a character
21688 of class **graph** in the program's current locale; see the Base Definitions volume of
21689 IEEE Std 1003.1-200x, Chapter 7, Locale.

21690 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character |
21691 code corresponding to a valid character in the current locale, or equal to the value of the macro
21692 WEOF. If the argument has any other value, the behavior is undefined.

21693 **RETURN VALUE**

21694 The *iswgraph*( ) function shall return non-zero if *wc* is a wide-character code with a visible
21695 representation; otherwise, it shall return 0.

21696 **ERRORS**

21697 No errors are defined.

21698 **EXAMPLES**

21699 None.

21700 **APPLICATION USAGE**

21701 To ensure applications portability, especially across natural languages, only this function and
21702 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21703 **RATIONALE**

21704 None.

21705 **FUTURE DIRECTIONS**

21706 None.

21707 **SEE ALSO**

21708 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
21709 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
21710 IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of
21711 IEEE Std 1003.1-200x, Chapter 7, Locale

21712 **CHANGE HISTORY**

21713 First released in Issue 4.

21714 **Issue 5**

21715 The following change has been made in this issue for alignment with
21716 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

21717 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21718 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

iswlower — test for a lowercase letter wide-character code

SYNOPSIS

```
#include <wctype.h>

int iswlower(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswlower*( ) function shall test whether *wc* is a wide-character code representing a character of class **lower** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

RETURN VALUE

The *iswlower*( ) function shall return non-zero if *wc* is a lowercase letter wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

21762 **Issue 6**

21763 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

21764 **NAME**

21765 iswprint — test for a printable wide-character code

21766 **SYNOPSIS**

21767
```
#include <wctype.h>
```

21768
```
int iswprint(wint_t wc);
```

21769 **DESCRIPTION**

21770 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21771 conflict between the requirements described here and the ISO C standard is unintentional. This
21772 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21773 The *iswprint*( ) function shall test whether *wc* is a wide-character code representing a character of
21774 class **print** in the program's current locale; see the Base Definitions volume of
21775 IEEE Std 1003.1-200x, Chapter 7, Locale.

21776 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
21777 code corresponding to a valid character in the current locale, or equal to the value of the macro
21778 WEOF. If the argument has any other value, the behavior is undefined.

21779 **RETURN VALUE**

21780 The *iswprint*( ) function shall return non-zero if *wc* is a printable wide-character code; otherwise,
21781 it shall return 0.

21782 **ERRORS**

21783 No errors are defined.

21784 **EXAMPLES**

21785 None.

21786 **APPLICATION USAGE**

21787 To ensure applications portability, especially across natural languages, only this function and
21788 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21789 **RATIONALE**

21790 None.

21791 **FUTURE DIRECTIONS**

21792 None.

21793 **SEE ALSO**

21794 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswpunct*( ),
21795 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
21796 IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of
21797 IEEE Std 1003.1-200x, Chapter 7, Locale

21798 **CHANGE HISTORY**

21799 First released in Issue 4.

21800 **Issue 5**

21801 The following change has been made in this issue for alignment with
21802 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

21803 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21804 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

21807 **NAME**

21808 iswpunct — test for a punctuation wide-character code

21809 **SYNOPSIS**

21810 `#include <wctype.h>`

21811 `int iswpunct(wint_t wc);`

21812 **DESCRIPTION**

21813 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21814 conflict between the requirements described here and the ISO C standard is unintentional. This
21815 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21816 The *iswpunct*( ) function shall test whether *wc* is a wide-character code representing a character
21817 of class **punct** in the program's current locale; see the Base Definitions volume of
21818 IEEE Std 1003.1-200x, Chapter 7, Locale.

21819 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character |
21820 code corresponding to a valid character in the current locale, or equal to the value of the macro
21821 WEOF. If the argument has any other value, the behavior is undefined.

21822 **RETURN VALUE**

21823 The *iswpunct*( ) function shall return non-zero if *wc* is a punctuation wide-character code;
21824 otherwise, it shall return 0.

21825 **ERRORS**

21826 No errors are defined.

21827 **EXAMPLES**

21828 None.

21829 **APPLICATION USAGE**

21830 To ensure applications portability, especially across natural languages, only this function and
21831 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21832 **RATIONALE**

21833 None.

21834 **FUTURE DIRECTIONS**

21835 None.

21836 **SEE ALSO**

21837 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ),
21838 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
21839 IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of
21840 IEEE Std 1003.1-200x, Chapter 7, Locale

21841 **CHANGE HISTORY**

21842 First released in Issue 4.

21843 **Issue 5**

21844 The following change has been made in this issue for alignment with
21845 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

21846 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21847 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

21850 **NAME**

21851 iswspace — test for a white-space wide-character code

21852 **SYNOPSIS**

21853 `#include <wctype.h>`

21854 `int iswspace(wint_t wc);`

21855 **DESCRIPTION**

21856 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21857 conflict between the requirements described here and the ISO C standard is unintentional. This
21858 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21859 The *iswspace*( ) function shall test whether *wc* is a wide-character code representing a character of
21860 class **space** in the program's current locale; see the Base Definitions volume of
21861 IEEE Std 1003.1-200x, Chapter 7, Locale.

21862 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character |
21863 code corresponding to a valid character in the current locale, or equal to the value of the macro
21864 WEOF. If the argument has any other value, the behavior is undefined.

21865 **RETURN VALUE**

21866 The *iswspace*( ) function shall return non-zero if *wc* is a white-space wide-character code;
21867 otherwise, it shall return 0.

21868 **ERRORS**

21869 No errors are defined.

21870 **EXAMPLES**

21871 None.

21872 **APPLICATION USAGE**

21873 To ensure applications portability, especially across natural languages, only this function and
21874 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21875 **RATIONALE**

21876 None.

21877 **FUTURE DIRECTIONS**

21878 None.

21879 **SEE ALSO**

21880 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ),
21881 *iswpunct*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
21882 IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of
21883 IEEE Std 1003.1-200x, Chapter 7, Locale

21884 **CHANGE HISTORY**

21885 First released in Issue 4.

21886 **Issue 5**

21887 The following change has been made in this issue for alignment with
21888 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

21889 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21890 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

21891 **Issue 6**
21892 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

iswupper — test for an uppercase letter wide-character code

SYNOPSIS

```
#include <wctype.h>

int iswupper(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswupper*( ) function shall test whether *wc* is a wide-character code representing a character of class **upper** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

RETURN VALUE

The *iswupper*( ) function shall return non-zero if *wc* is an uppercase letter wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

21936 **NAME**

21937 iswxdigit — test for a hexadecimal digit wide-character code

21938 **SYNOPSIS**

21939 `#include <wctype.h>`

21940 `int iswxdigit(wint_t wc);`

21941 **DESCRIPTION**

21942 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21943 conflict between the requirements described here and the ISO C standard is unintentional. This
21944 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21945 The *iswxdigit*( ) function shall test whether *wc* is a wide-character code representing a character
21946 of class **xdigit** in the program's current locale; see the Base Definitions volume of
21947 IEEE Std 1003.1-200x, Chapter 7, Locale.

21948 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character |
21949 code corresponding to a valid character in the current locale, or equal to the value of the macro
21950 WEOF. If the argument has any other value, the behavior is undefined.

21951 **RETURN VALUE**

21952 The *iswxdigit*( ) function shall return non-zero if *wc* is a hexadecimal digit wide-character code;
21953 otherwise, it shall return 0.

21954 **ERRORS**

21955 No errors are defined.

21956 **EXAMPLES**

21957 None.

21958 **APPLICATION USAGE**

21959 To ensure applications portability, especially across natural languages, only this function and
21960 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21961 **RATIONALE**

21962 None.

21963 **FUTURE DIRECTIONS**

21964 None.

21965 **SEE ALSO**

21966 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ),
21967 *iswpunct*( ), *iswspace*( ), *iswupper*( ), *setlocale*( ), the Base Definitions volume of
21968 IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**

21969 **CHANGE HISTORY**

21970 First released in Issue 4.

21971 **Issue 5**

21972 The following change has been made in this issue for alignment with
21973 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

21974 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21975 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

21976 **Issue 6**

21977 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

21978 **NAME**

21979 isxdigit — test for a hexadecimal digit

21980 **SYNOPSIS**

21981 `#include <ctype.h>`

21982 `int isxdigit(int c);`

21983 **DESCRIPTION**

21984 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21985 conflict between the requirements described here and the ISO C standard is unintentional. This
21986 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21987 The *isxdigit*( ) function shall test whether *c* is a character of class **xdigit** in the program's current
21988 locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21989 The *c* argument is an **int**, the value of which the application shall ensure is a character |
21990 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21991 any other value, the behavior is undefined.

21992 **RETURN VALUE**

21993 The *isxdigit*( ) function shall return non-zero if *c* is a hexadecimal digit; otherwise, it shall return
21994 0.

21995 **ERRORS**

21996 No errors are defined.

21997 **EXAMPLES**

21998 None.

21999 **APPLICATION USAGE**

22000 To ensure applications portability, especially across natural languages, only this function and
22001 those listed in the SEE ALSO section should be used for character classification.

22002 **RATIONALE**

22003 None.

22004 **FUTURE DIRECTIONS**

22005 None.

22006 **SEE ALSO**

22007 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
22008 the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**

22009 **CHANGE HISTORY**

22010 First released in Issue 1. Derived from Issue 1 of the SVID.

22011 **Issue 6**

22012 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

22013 **NAME**

22014 j0, j1, jn — Bessel functions of the first kind

22015 **SYNOPSIS**

22016 XSI
```
#include <math.h>

double j0(double x);
double j1(double x);
double jn(int n, double x);
```
22017
22018
22019
22020

22021 **DESCRIPTION**

22022 The *j0*( ), *j1*( ), and *jn*( ) functions shall compute Bessel functions of *x* of the first kind of orders 0,
22023 1, and *n*, respectively.

22024 An application wishing to check for error situations should set *errno* to zero and call
22025 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
22026 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
22027 zero, an error has occurred.

22028 **RETURN VALUE**

22029 Upon successful completion, these functions shall return the relevant Bessel value of *x* of the
22030 first kind.

22031 If the *x* argument is too large in magnitude, or the correct result would cause underflow, 0 shall
22032 be returned and a range error may occur.

22033 If *x* is NaN, a NaN shall be returned.

22034 **ERRORS**

22035 These functions may fail if:

22036 Range Error — The value of *x* was too large in magnitude, or an underflow occurred.

22037 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
22038 then *errno* shall be set to [ERANGE]. If the integer expression |
22039 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow |
22040 floating-point exception shall be raised. |

22041 No other errors shall occur.

22042 **EXAMPLES**

22043 None.

22044 **APPLICATION USAGE**

22045 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
22046 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

22047 **RATIONALE**

22048 None.

22049 **FUTURE DIRECTIONS**

22050 None.

22051 **SEE ALSO**

22052 *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *y0*( ), the Base Definitions volume of IEEE Std 1003.1-200x, |
22053 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

22054 **CHANGE HISTORY**

22055 First released in Issue 1. Derived from Issue 1 of the SVID.

22056 **Issue 5**

22057 The DESCRIPTION is updated to indicate how an application should check for an error. This
22058 text was previously published in the APPLICATION USAGE section.

22059 **Issue 6**

22060 The may fail [EDOM] error is removed for the case for NaN.

22061 The RETURN VALUE and ERRORS sections are reworked for alignment of the error handling |
22062 with the ISO / IEC 9899: 1999 standard.

22063 **NAME**

22064 jrand48 — generate a uniformly distributed pseudo-random long signed integer

22065 **SYNOPSIS**

22066 XSI
```
#include <stdlib.h>

long jrand48(unsigned short xsubi[3]);
```
22067 |

22068 |

22069 **DESCRIPTION**

22070 Refer to *drand48*( ).

22071 **NAME**

22072 kill — send a signal to a process or a group of processes

22073 **SYNOPSIS**

22074 CX
```
#include <signal.h>

int kill(pid_t pid, int sig);
```
22075

22076

22077 **DESCRIPTION**

22078 The *kill*( ) function shall send a signal to a process or a group of processes specified by *pid*. The
22079 signal to be sent is specified by *sig* and is either one from the list given in **<signal.h>** or 0. If *sig* is
22080 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can
22081 be used to check the validity of *pid*.

22082 For a process to have permission to send a signal to a process designated by *pid*, unless the
22083 sending process has appropriate privileges, the real or effective user ID of the sending process
22084 shall match the real or saved set-user-ID of the receiving process.

22085 If *pid* is greater than 0, *sig* shall be sent to the process whose process ID is equal to *pid*.

22086 If *pid* is 0, *sig* shall be sent to all processes (excluding an unspecified set of system processes)
22087 whose process group ID is equal to the process group ID of the sender, and for which the
22088 process has permission to send a signal.

22089 If *pid* is –1, *sig* shall be sent to all processes (excluding an unspecified set of system processes) for
22090 which the process has permission to send that signal.

22091 If *pid* is negative, but not –1, *sig* shall be sent to all processes (excluding an unspecified set of
22092 system processes) whose process group ID is equal to the absolute value of *pid*, and for which
22093 the process has permission to send a signal.

22094 If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* is not blocked for
22095 the calling thread and if no other thread has *sig* unblocked or is waiting in a *sigwait*( ) function
22096 for *sig*, either *sig* or at least one pending unblocked signal shall be delivered to the sending
22097 thread before *kill*( ) returns.

22098 The user ID tests described above shall not be applied when sending SIGCONT to a process that
22099 is a member of the same session as the sending process.

22100 An implementation that provides extended security controls may impose further
22101 implementation-defined restrictions on the sending of signals, including the null signal. In
22102 particular, the system may deny the existence of some or all of the processes specified by *pid*.

22103 The *kill*( ) function is successful if the process has permission to send *sig* to any of the processes
22104 specified by *pid*. If *kill*( ) fails, no signal shall be sent.

22105 **RETURN VALUE**

22106 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
22107 indicate the error.

22108 **ERRORS**

22109 The *kill*( ) function shall fail if:

22110 [EINVAL] The value of the *sig* argument is an invalid or unsupported signal number.

22111 [EPERM] The process does not have permission to send the signal to any receiving
22112 process.

22113 [ESRCH] No process or process group can be found corresponding to that specified by
22114 *pid*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The semantics for permission checking for *kill*( ) differed between System V and most other implementations, such as Version 7 or 4.3 BSD. The semantics chosen for this volume of IEEE Std 1003.1-200x agree with System V. Specifically, a set-user-ID process cannot protect itself against signals (or at least not against SIGKILL) unless it changes its real user ID. This choice allows the user who starts an application to send it signals even if it changes its effective user ID. The other semantics give more power to an application that wants to protect itself from the user who ran it.

Some implementations provide semantic extensions to the *kill*( ) function when the absolute value of *pid* is greater than some maximum, or otherwise special, value. Negative values are a flag to *kill*( ). Since most implementations return [ESRCH] in this case, this behavior is not included in this volume of IEEE Std 1003.1-200x, although a conforming implementation could provide such an extension.

The implementation-defined processes to which a signal cannot be sent may include the scheduler or *init*.

There was initially strong sentiment to specify that, if *pid* specifies that a signal be sent to the calling process and that signal is not blocked, that signal would be delivered before *kill*( ) returns. This would permit a process to call *kill*( ) and be guaranteed that the call never return. However, historical implementations that provide only the *signal*( ) function make only the weaker guarantee in this volume of IEEE Std 1003.1-200x, because they only deliver one signal each time a process enters the kernel. Modifications to such implementations to support the *sigaction*( ) function generally require entry to the kernel following return from a signal-catching function, in order to restore the signal mask. Such modifications have the effect of satisfying the stronger requirement, at least when *sigaction*( ) is used, but not necessarily when *signal*( ) is used. The developers of this volume of IEEE Std 1003.1-200x considered making the stronger requirement except when *signal*( ) is used, but felt this would be unnecessarily complex. Implementors are encouraged to meet the stronger requirement whenever possible. In practice, the weaker requirement is the same, except in the rare case when two signals arrive during a very short window. This reasoning also applies to a similar requirement for *sigprocmask*( ).

In 4.2 BSD, the SIGCONT signal can be sent to any descendant process regardless of user-ID security checks. This allows a job control shell to continue a job even if processes in the job have altered their user IDs (as in the *su* command). In keeping with the addition of the concept of sessions, similar functionality is provided by allowing the SIGCONT signal to be sent to any process in the same session regardless of user ID security checks. This is less restrictive than BSD in the sense that ancestor processes (in the same session) can now be the recipient. It is more restrictive than BSD in the sense that descendant processes that form new sessions are now subject to the user ID checks. A similar relaxation of security is not necessary for the other job control signals since those signals are typically sent by the terminal driver in recognition of special characters being typed; the terminal driver bypasses all security checks.

In secure implementations, a process may be restricted from sending a signal to a process having a different security label. In order to prevent the existence or nonexistence of a process from being used as a covert channel, such processes should appear nonexistent to the sender; that is, [ESRCH] should be returned, rather than [EPERM], if *pid* refers only to such processes.

Existing implementations vary on the result of a *kill*( ) with *pid* indicating an inactive process (a terminated process that has not been waited for by its parent). Some indicate success on such a call (subject to permission checking), while others give an error of [ESRCH]. Since the definition of process lifetime in this volume of IEEE Std 1003.1-200x covers inactive processes, the [ESRCH] error as described is inappropriate in this case. In particular, this means that an application cannot have a parent process check for termination of a particular child with *kill*( ). (Usually this is done with the null signal; this can be done reliably with *waitpid*( ).)

There is some belief that the name *kill*( ) is misleading, since the function is not always intended to cause process termination. However, the name is common to all historical implementations, and any change would be in conflict with the goal of minimal changes to existing application code.

FUTURE DIRECTIONS

None.

SEE ALSO

getpid( ), *raise*( ), *setsid*( ), *sigaction*( ), *sigqueue*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with POSIX Threads Extension.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the second paragraph is reworded to indicate that the saved set-user-ID of the calling process is checked in place of its effective user ID. This is a FIPS requirement.
- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The behavior when *pid* is –1 is now specified. It was previously explicitly unspecified in the POSIX.1-1988 standard.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

22195 **NAME**

22196 killpg — send a signal to a process group

22197 **SYNOPSIS**

22198 XSI
```
#include <signal.h>

int killpg(pid_t pgrp, int sig);
```
22200

22201 **DESCRIPTION**

22202 The *killpg*( ) function shall send the signal specified by *sig* to the process group specified by *pgrp*.

22203 If *pgrp* is greater than 1, *killpg*(*pgrp*, *sig*) shall be equivalent to *kill*(–*pgrp*, *sig*). If *pgrp* is less than or
22204 equal to 1, the behavior of *killpg*( ) is undefined.

22205 **RETURN VALUE**

22206 Refer to *kill*( ).

22207 **ERRORS**

22208 Refer to *kill*( ).

22209 **EXAMPLES**

22210 None.

22211 **APPLICATION USAGE**

22212 None.

22213 **RATIONALE**

22214 None.

22215 **FUTURE DIRECTIONS**

22216 None.

22217 **SEE ALSO**

22218 *getpgid*( ), *getpid*( ), *kill*( ), *raise*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

22219 **CHANGE HISTORY**

22220 First released in Issue 4, Version 2.

22221 **Issue 5**

22222 Moved from X/OPEN UNIX extension to BASE.

22223 **NAME**

22224 l64a — convert a 32-bit integer to a radix-64 ASCII string

22225 **SYNOPSIS**

22226 XSI
```
#include <stdlib.h>

char *l64a(long value);
```
22227

22228

22229 **DESCRIPTION**

22230 Refer to *a64l*( ).

22231 **NAME**

22232 labs, llabs — return a long integer absolute value

22233 **SYNOPSIS**

22234
```
#include <stdlib.h>
```

22235
```
long labs(long i);
```
22236
```
long long llabs(long long i);
```

22237 **DESCRIPTION**

22238 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22239 conflict between the requirements described here and the ISO C standard is unintentional. This
22240 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22241 The *labs*( ) function shall compute the absolute value of the **long** integer operand *i*. The *llabs*( )
22242 function shall compute the absolute value of the **long long** integer operand *i*. If the result cannot
22243 be represented, the behavior is undefined.

22244 **RETURN VALUE**

22245 The *labs*( ) function shall return the absolute value of the **long** integer operand. The *labs*( )
22246 function shall return the absolute value of the **long long** integer operand.

22247 **ERRORS**

22248 No errors are defined.

22249 **EXAMPLES**

22250 None.

22251 **APPLICATION USAGE**

22252 None.

22253 **RATIONALE**

22254 None.

22255 **FUTURE DIRECTIONS**

22256 None.

22257 **SEE ALSO**

22258 *abs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

22259 **CHANGE HISTORY**

22260 First released in Issue 4. Derived from the ISO C standard.

22261 **Issue 6**

22262 The *llabs*( ) function is added for alignment with the ISO / IEC 9899: 1999 standard.

22263 **NAME**

22264 lchown — change the owner and group of a symbolic link

22265 **SYNOPSIS**

22266 XSI
```
#include <unistd.h>

int lchown(const char *path, uid_t owner, gid_t group);
```
22268

22269 **DESCRIPTION**

22270 The *lchown*( ) function shall be equivalent to *chown*( ), except in the case where the named file is a |
22271 symbolic link. In this case, *lchown*( ) shall change the ownership of the symbolic link file itself,
22272 while *chown*( ) changes the ownership of the file or directory to which the symbolic link refers.

22273 **RETURN VALUE**

22274 Upon successful completion, *lchown*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to
22275 indicate an error.

22276 **ERRORS**

22277 The *lchown*( ) function shall fail if:

22278 [EACCES] Search permission is denied on a component of the path prefix of *path*.

22279 [EINVAL] The owner or group ID is not a value supported by the implementation.

22280 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
22281 argument.

22282 [ENAMETOOLONG]
22283 The length of a pathname exceeds {PATH_MAX} or a pathname component is |
22284 longer than {NAME_MAX}. | |

22285 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

22286 [ENOTDIR] A component of the path prefix of *path* is not a directory.

22287 [EOPNOTSUPP] The *path* argument names a symbolic link and the implementation does not
22288 support setting the owner or group of a symbolic link.

22289 [EPERM] The effective user ID does not match the owner of the file and the process
22290 does not have appropriate privileges.

22291 [EROFS] The file resides on a read-only file system.

22292 The *lchown*( ) function may fail if:

22293 [EIO] An I/O error occurred while reading or writing to the file system.

22294 [EINTR] A signal was caught during execution of the function.

22295 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
22296 resolution of the *path* argument.

22297 [ENAMETOOLONG]
22298 Pathname resolution of a symbolic link produced an intermediate result |
22299 whose length exceeds {PATH_MAX}.

EXAMPLES

Changing the Current Owner of a File

The following example shows how to change the ownership of the symbolic link named **/modules/pass1** to the user ID associated with ‘‘jones’’ and the group ID associated with ‘‘cnd’’.

The numeric value for the user ID is obtained by using the *getpwnam*( ) function. The numeric value for the group ID is obtained by using the *getgrnam*( ) function.

```
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>

struct passwd *pwd;
struct group  *grp;
char          *path = "/modules/pass1";
...
pwd = getpwnam("jones");
grp = getgrnam("cnd");
lchown(path, pwd->pw_uid, grp->gr_gid);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chown( ), *symlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

22332 **NAME**

22333 lcong48 — seed a uniformly distributed pseudo-random signed long integer generator

22334 **SYNOPSIS**

22335 XSI
```
#include <stdlib.h>
```

22336
```
void lcong48(unsigned short param[7]);
```

22337

22338 **DESCRIPTION**

22339 Refer to *drand48*( ).

22340 **NAME**

22341 ldexp, ldexpf, ldexpl — load exponent of a floating-point number

22342 **SYNOPSIS**

22343 `#include <math.h>`

22344 `double ldexp(double x, int exp);`
22345 `float ldexpf(float x, int exp);`
22346 `long double ldexpl(long double x, int exp);`

22347 **DESCRIPTION**

22348 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22349 conflict between the requirements described here and the ISO C standard is unintentional. This
22350 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22351 These functions shall compute the quantity $x * 2^{exp}$.

22352 An application wishing to check for error situations should set *errno* to zero and call
22353 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
22354 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
22355 zero, an error has occurred.

22356 **RETURN VALUE**

22357 Upon successful completion, these functions shall return *x* multiplied by 2, raised to the power
22358 *exp*.

22359 If these functions would cause overflow, a range error shall occur and *ldexp*( ), *ldexpf*( ), and
22360 *ldexpl*( ) shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (according to the sign of
22361 *x*), respectively.

22362 If the correct value would cause underflow, and is not representable, a range error may occur,
22363 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

22364 MX If *x* is NaN, a NaN shall be returned.

22365 If *x* is ±0 or ±Inf, *x* shall be returned.

22366 If *exp* is 0, *x* shall be returned.

22367 If the correct value would cause underflow, and is representable, a range error may occur and
22368 the correct value shall be returned.

22369 **ERRORS**

22370 These functions shall fail if:

22371 Range Error The result overflows.

22372 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
22373 then *errno* shall be set to [ERANGE]. If the integer expression |
22374 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow |
22375 floating-point exception shall be raised. |

22376 These functions may fail if:

22377 Range Error The result underflows.

22378 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
22379 then *errno* shall be set to [ERANGE]. If the integer expression |
22380 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow |
22381 floating-point exception shall be raised. |

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *frexp*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *ldexpf*( ) and *ldexpl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

22407 **NAME**

22408 ldiv, lldiv — compute quotient and remainder of a long division

22409 **SYNOPSIS**

22410
```
#include <stdlib.h>
```

22411
```
ldiv_t ldiv(long numer, long denom);
```
22412
```
lldiv_t lldiv(long long numer, long long denom);
```

22413 **DESCRIPTION**

22414 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22415 conflict between the requirements described here and the ISO C standard is unintentional. This
22416 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22417 These functions shall compute the quotient and remainder of the division of the numerator
22418 *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the **long**
22419 integer (for the *ldiv*( ) function) or **long long** integer (for the *lldiv*( ) function) of lesser magnitude
22420 that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is
22421 undefined; otherwise, *quot* * *denom*+*rem* shall equal *numer*.

22422 **RETURN VALUE**

22423 The *ldiv*( ) function shall return a structure of type **ldiv_t**, comprising both the quotient and the |
22424 remainder. The structure shall include the following members, in any order: |

22425
```
long   quot;    /* Quotient */
```
22426
```
long   rem;     /* Remainder */
```

22427 The *lldiv*( ) function shall return a structure of type **lldiv_t**, comprising both the quotient and the |
22428 remainder. The structure shall include the following members, in any order: |

22429
```
long long   quot;    /* Quotient */
```
22430
```
long long   rem;     /* Remainder */
```

22431 **ERRORS**

22432 No errors are defined.

22433 **EXAMPLES**

22434 None.

22435 **APPLICATION USAGE**

22436 None.

22437 **RATIONALE**

22438 None.

22439 **FUTURE DIRECTIONS**

22440 None.

22441 **SEE ALSO**

22442 *div*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

22443 **CHANGE HISTORY**

22444 First released in Issue 4. Derived from the ISO C standard.

22445 **Issue 6**

22446 The *lldiv*( ) function is added for alignment with the ISO / IEC 9899: 1999 standard.

22447 **NAME**

22448 lfind — find entry in a linear search table

22449 **SYNOPSIS**

22450 XSI
```
#include <search.h>

void *lfind(const void *key, const void *base, size_t *nelp,
    size_t width, int (*compar)(const void *, const void *));
```
22451

22452

22453

22454 **DESCRIPTION**

22455 Refer to *lsearch*( ).

22456 **NAME**

22457 lgamma, lgammaf, lgammal — log gamma function

22458 **SYNOPSIS**

22459 `#include <math.h>`

22460 `double lgamma(double x);`
22461 `float lgammaf(float x);`
22462 `long double lgammal(long double x);`
22463 XSI `extern int signgam;`
22464

22465 **DESCRIPTION**

22466 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22467 conflict between the requirements described here and the ISO C standard is unintentional. This
22468 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22469 These functions shall compute $\log_e|\Gamma(x)|$ where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t}t^{x-1}dt$. The argument x
22470 need not be a non-positive integer ($\Gamma(x)$ is defined over the reals, except the non-positive
22471 integers).

22472 XSI The sign of $\Gamma(x)$ is returned in the external integer *signgam*.

22473 CX These functions need not be reentrant. A function that is not required to be reentrant is not
22474 required to be thread-safe.

22475 An application wishing to check for error situations should set *errno* to zero and call
22476 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
22477 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
22478 zero, an error has occurred.

22479 **RETURN VALUE**

22480 Upon successful completion, these functions shall return the logarithmic gamma of *x*.

22481 If *x* is a non-positive integer, a pole error shall occur and *lgamma*( ), *lgammaf*( ), and *lgammal*( )
22482 shall return +HUGE_VAL, +HUGE_VALF, and +HUGE_VALL, respectively.

22483 If the correct value would cause overflow, a range error shall occur and *lgamma*( ), *lgammaf*( ),
22484 and *lgammal*( ) shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (having the same
22485 sign as the correct value), respectively.

22486 MX If *x* is NaN, a NaN shall be returned.

22487 If *x* is 1 or 2, +0 shall be returned.

22488 If *x* is ±Inf, +Inf shall be returned

22489 **ERRORS**

22490 These functions shall fail if:

22491 Pole Error — The *x* argument is a negative integer or zero.

22492 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised.

22493 Range Error — The result overflows

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exp( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *lgamma*( ) function is no longer marked as an extension.

The *lgammaf*( ) and *lgammal*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

XSI extensions are marked.

22529 **NAME**

22530 link — link to a file

22531 **SYNOPSIS**

22532 `#include <unistd.h>`

22533 `int link(const char *path1, const char *path2);`

22534 **DESCRIPTION**

22535 The *link*( ) function shall create a new link (directory entry) for the existing file, *path1*.

22536 The *path1* argument points to a pathname naming an existing file. The *path2* argument points to |
22537 a pathname naming the new directory entry to be created. The *link*( ) function shall atomically |
22538 create a new link for the existing file and the link count of the file shall be incremented by one.

22539 If *path1* names a directory, *link*( ) shall fail unless the process has appropriate privileges and the
22540 implementation supports using *link*( ) on directories.

22541 Upon successful completion, *link*( ) shall mark for update the *st_ctime* field of the file. Also, the
22542 *st_ctime* and *st_mtime* fields of the directory that contains the new entry shall be marked for
22543 update.

22544 If *link*( ) fails, no link shall be created and the link count of the file shall remain unchanged.

22545 The implementation may require that the calling process has permission to access the existing
22546 file.

22547 **RETURN VALUE**

22548 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
22549 indicate the error.

22550 **ERRORS**

22551 The *link*( ) function shall fail if:

22552 [EACCES] A component of either path prefix denies search permission, or the requested
22553 link requires writing in a directory that denies write permission, or the calling
22554 process does not have permission to access the existing file and this is
22555 required by the implementation.

22556 [EEXIST] The *path2* argument resolves to an existing file or refers to a symbolic link.

22557 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path1* or
22558 *path2* argument.

22559 [EMLINK] The number of links to the file named by *path1* would exceed {LINK_MAX}.

22560 [ENAMETOOLONG]
22561 The length of the *path1* or *path2* argument exceeds {PATH_MAX} or a |
22562 pathname component is longer than {NAME_MAX}. |

22563 [ENOENT] A component of either path prefix does not exist; the file named by *path1* does
22564 not exist; or *path1* or *path2* points to an empty string.

22565 [ENOSPC] The directory to contain the link cannot be extended.

22566 [ENOTDIR] A component of either path prefix is not a directory.

22567 [EPERM] The file named by *path1* is a directory and either the calling process does not
22568 have appropriate privileges or the implementation prohibits using *link*( ) on
22569 directories.

[EROFS] The requested link requires writing in a directory on a read-only file system.

[EXDEV] The link named by *path2* and the file named by *path1* are on different file systems and the implementation does not support links between file systems, or *path1* refers to a named STREAM. XSR

The *link*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path1* or *path2* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path1* or *path2* argument, the length of the substituted pathname string exceeded {PATH_MAX}. |

EXAMPLES

Creating a Link to a File

The following example shows how to create a link to a file named **/home/cnd/mod1** by creating a new directory entry named **/modules/pass1**.

```
#include <unistd.h>

char *path1 = "/home/cnd/mod1";
char *path2 = "/modules/pass1";
int   status;
...
status = link (path1, path2);
```

Creating a Link to a File Within a Program

In the following program example, the *link*( ) function links the **/etc/passwd** file (defined as **PASSWDFILE**) to a file named **/etc/opasswd** (defined as **SAVEFILE**), which is used to save the current password file. Then, after removing the current password file (defined as **PASSWDFILE**), the new password file is saved as the current password file using the *link*( ) function again. |

```
#include <unistd.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
#define SAVEFILE "/etc/opasswd"
...
/* Save current password file */
link (PASSWDFILE, SAVEFILE);

/* Remove current password file. */
unlink (PASSWDFILE);

/* Save new password file as current password file. */
link (LOCKFILE,PASSWDFILE);
```

APPLICATION USAGE

Some implementations do allow links between file systems.

RATIONALE

Linking to a directory is restricted to the superuser in most historical implementations because this capability may produce loops in the file hierarchy or otherwise corrupt the file system. This volume of IEEE Std 1003.1-200x continues that philosophy by prohibiting *link*() and *unlink*() from doing this. Other functions could do it if the implementor designed such an extension.

Some historical implementations allow linking of files on different file systems. Wording was added to explicitly allow this optional behavior. |

The exception for cross-file system links is intended to apply only to links that are programmatically indistinguishable from ''hard'' links.

FUTURE DIRECTIONS

None.

SEE ALSO

symlink(), *unlink*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- An explanation is added of action when *path2* refers to a symbolic link.
- The [ELOOP] optional error condition is added.

22633 **NAME**

22634 lio_listio — list directed I/O (**REALTIME**)

22635 **SYNOPSIS**

22636 AIO
```
#include <aio.h>

int lio_listio(int mode, struct aiocb *restrict const list[restrict],
    int nent, struct sigevent *restrict sig);
```
22639

22640 **DESCRIPTION**

22641 The *lio_listio*( ) function shall initiate a list of I/O requests with a single function call.

22642 The *mode* argument takes one of the values LIO_WAIT or LIO_NOWAIT declared in **<aio.h>** and
22643 determines whether the function returns when the I/O operations have been completed, or as
22644 soon as the operations have been queued. If the *mode* argument is LIO_WAIT, the function shall
22645 wait until all I/O is complete and the *sig* argument shall be ignored.

22646 If the *mode* argument is LIO_NOWAIT, the function shall return immediately, and asynchronous
22647 notification shall occur, according to the *sig* argument, when all the I/O operations complete. If
22648 *sig* is NULL, then no asynchronous notification shall occur. If *sig* is not NULL, asynchronous
22649 notification occurs as specified in Section 2.4.1 (on page 478) when all the requests in *list* have
22650 completed.

22651 The I/O requests enumerated by *list* are submitted in an unspecified order.

22652 The *list* argument is an array of pointers to **aiocb** structures. The array contains *nent* elements.
22653 The array may contain NULL elements, which shall be ignored.

22654 The *aio_lio_opcode* field of each **aiocb** structure specifies the operation to be performed. The
22655 supported operations are LIO_READ, LIO_WRITE, and LIO_NOP; these symbols are defined in
22656 **<aio.h>**. The LIO_NOP operation causes the list entry to be ignored. If the *aio_lio_opcode*
22657 element is equal to LIO_READ, then an I/O operation is submitted as if by a call to *aio_read*( )
22658 with the *aiocbp* equal to the address of the **aiocb** structure. If the *aio_lio_opcode* element is equal
22659 to LIO_WRITE, then an I/O operation is submitted as if by a call to *aio_write*( ) with the *aiocbp*
22660 equal to the address of the **aiocb** structure.

22661 The *aio_fildes* member specifies the file descriptor on which the operation is to be performed.

22662 The *aio_buf* member specifies the address of the buffer to or from which the data is transferred.

22663 The *aio_nbytes* member specifies the number of bytes of data to be transferred.

22664 The members of the **aiocb** structure further describe the I/O operation to be performed, in a
22665 manner identical to that of the corresponding **aiocb** structure when used by the *aio_read*( ) and
22666 *aio_write*( ) functions.

22667 The *nent* argument specifies how many elements are members of the list; that is, the length of the
22668 array.

22669 The behavior of this function is altered according to the definitions of synchronized I/O data
22670 integrity completion and synchronized I/O file integrity completion if synchronized I/O is
22671 enabled on the file associated with *aio_fildes*.

22672 For regular files, no data transfer shall occur past the offset maximum established in the open
22673 file description associated with *aiocbp->aio_fildes*.

22674 **RETURN VALUE**

22675 If the *mode* argument has the value LIO_NOWAIT, the *lio_listio*( ) function shall return the value
22676 zero if the I/O operations are successfully queued; otherwise, the function shall return the value
22677 –1 and set *errno* to indicate the error.

22678 If the *mode* argument has the value LIO_WAIT, the *lio_listio*( ) function shall return the value
22679 zero when all the indicated I/O has completed successfully. Otherwise, *lio_listio*( ) shall return a
22680 value of –1 and set *errno* to indicate the error.

22681 In either case, the return value only indicates the success or failure of the *lio_listio*( ) call itself,
22682 not the status of the individual I/O requests. In some cases one or more of the I/O requests
22683 contained in the list may fail. Failure of an individual request does not prevent completion of
22684 any other individual request. To determine the outcome of each I/O request, the application
22685 shall examine the error status associated with each **aiocb** control block. The error statuses so
22686 returned are identical to those returned as the result of an *aio_read*( ) or *aio_write*( ) function.

22687 **ERRORS**

22688 The *lio_listio*( ) function shall fail if:

22689 [EAGAIN] The resources necessary to queue all the I/O requests were not available. The
22690 application may check the error status for each **aiocb** to determine the
22691 individual request(s) that failed.

22692 [EAGAIN] The number of entries indicated by *nent* would cause the system-wide limit
22693 {AIO_MAX} to be exceeded.

22694 [EINVAL] The *mode* argument is not a proper value, or the value of *nent* was greater than
22695 {AIO_LISTIO_MAX}.

22696 [EINTR] A signal was delivered while waiting for all I/O requests to complete during
22697 an LIO_WAIT operation. Note that, since each I/O operation invoked by
22698 *lio_listio*( ) may possibly provoke a signal when it completes, this error return
22699 may be caused by the completion of one (or more) of the very I/O operations
22700 being awaited. Outstanding I/O requests are not canceled, and the application
22701 shall examine each list element to determine whether the request was
22702 initiated, canceled, or completed.

22703 [EIO] One or more of the individual I/O operations failed. The application may
22704 check the error status for each **aiocb** structure to determine the individual
22705 request(s) that failed.

22706 In addition to the errors returned by the *lio_listio*( ) function, if the *lio_listio*( ) function succeeds
22707 or fails with errors of [EAGAIN], [EINTR], or [EIO], then some of the I/O specified by the list
22708 may have been initiated. If the *lio_listio*( ) function fails with an error code other than [EAGAIN],
22709 [EINTR], or [EIO], no operations from the list shall have been initiated. The I/O operation
22710 indicated by each list element can encounter errors specific to the individual read or write
22711 function being performed. In this event, the error status for each **aiocb** control block contains the
22712 associated error code. The error codes that can be set are the same as would be set by a *read*( ) or
22713 *write*( ) function, with the following additional error codes possible:

22714 [EAGAIN] The requested I/O operation was not queued due to resource limitations.

22715 [ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit
22716 *aio_cancel*( ) request.

22717 [EFBIG] The *aiocbp->aio_lio_opcode* is LIO_WRITE, the file is a regular file, *aiocbp-*
22718 *>aio_nbytes* is greater than 0, and the *aiocbp->aio_offset* is greater than or equal
22719 to the offset maximum in the open file description associated with *aiocbp-*

22720 *>aio_fildes*.

22721 [EINPROGRESS] The requested I/O is in progress.

22722 [EOVERFLOW] The *aiocbp->aio_lio_opcode* is LIO_READ, the file is a regular file, *aiocbp->aio_nbytes* is greater than 0, and the *aiocbp->aio_offset* is before the end-of-file and is greater than or equal to the offset maximum in the open file description associated with *aiocbp->aio_fildes*.

22726 **EXAMPLES**

22727 None.

22728 **APPLICATION USAGE**

22729 None.

22730 **RATIONALE**

22731 Although it may appear that there are inconsistencies in the specified circumstances for error codes, the [EIO] error condition applies when any circumstance relating to an individual operation makes that operation fail. This might be due to a badly formulated request (for example, the *aio_lio_opcode* field is invalid, and *aio_error*( ) returns [EINVAL]) or might arise from application behavior (for example, the file descriptor is closed before the operation is initiated, and *aio_error*( ) returns [EBADF]).

22737 The limitation on the set of error codes returned when operations from the list shall have been initiated enables applications to know when operations have been started and whether *aio_error*( ) is valid for a specific operation.

22740 **FUTURE DIRECTIONS**

22741 None.

22742 **SEE ALSO**

22743 *aio_read*( ), *aio_write*( ), *aio_error*( ), *aio_return*( ), *aio_cancel*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lseek*( ), *read*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

22745 **CHANGE HISTORY**

22746 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

22747 Large File Summit extensions are added.

22748 **Issue 6**

22749 The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

22751 The *lio_listio*( ) function is marked as part of the Asynchronous Input and Output option.

22752 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

22754 - In the DESCRIPTION, text is added to indicate that for regular files no data transfer occurs past the offset maximum established in the open file description associated with *aiocbp->aio_fildes*. This change is to support large files.

22757 - The [EBIG] and [EOVERFLOW] error conditions are defined. This change is to support large files.

22759 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

22760 The **restrict** keyword is added to the *lio_listio*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

22762 **NAME**

22763 listen — listen for socket connections and limit the queue of incoming connections

22764 **SYNOPSIS**

22765 `#include <sys/socket.h>`

22766 `int listen(int socket, int backlog);`

22767 **DESCRIPTION**

22768 The *listen*( ) function shall mark a connection-mode socket, specified by the *socket* argument, as
22769 accepting connections.

22770 The *backlog* argument provides a hint to the implementation which the implementation shall use
22771 to limit the number of outstanding connections in the socket's listen queue. Implementations
22772 may impose a limit on *backlog* and silently reduce the specified value. Normally, a larger *backlog*
22773 argument value shall result in a larger or equal length of the listen queue. Implementations shall
22774 support values of *backlog* up to SOMAXCONN, defined in **<sys/socket.h>**.

22775 The implementation may include incomplete connections in its listen queue. The limits on the
22776 number of incomplete connections and completed connections queued may be different.

22777 The implementation may have an upper limit on the length of the listen queue—either global or
22778 per accepting socket. If *backlog* exceeds this limit, the length of the listen queue is set to the limit.

22779 If *listen*( ) is called with a *backlog* argument value that is less than 0, the function behaves as if it
22780 had been called with a *backlog* argument value of 0.

22781 A *backlog* argument of 0 may allow the socket to accept connections, in which case the length of
22782 the listen queue may be set to an implementation-defined minimum value.

22783 The socket in use may require the process to have appropriate privileges to use the *listen*( )
22784 function.

22785 **RETURN VALUE**

22786 Upon successful completions, *listen*( ) shall return 0; otherwise, –1 shall be returned and *errno* set
22787 to indicate the error.

22788 **ERRORS**

22789 The *listen*( ) function shall fail if:

22790 [EBADF] The *socket* argument is not a valid file descriptor.

22791 [EDESTADDRREQ]
22792 The socket is not bound to a local address, and the protocol does not support
22793 listening on an unbound socket.

22794 [EINVAL] The *socket* is already connected.

22795 [ENOTSOCK] The *socket* argument does not refer to a socket.

22796 [EOPNOTSUPP] The socket protocol does not support *listen*( ).

22797 The *listen*( ) function may fail if:

22798 [EACCES] The calling process does not have the appropriate privileges.

22799 [EINVAL] The *socket* has been shut down.

22800 [ENOBUFS] Insufficient resources are available in the system to complete the call.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

accept( ), *connect*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The DESCRIPTION is updated to describe the relationship of SOMAXCONN and the *backlog* argument.

22815 **NAME**

22816 llabs — return a long integer absolute value

22817 **SYNOPSIS**

22818 `#include <stdlib.h>`

22819 `long long llabs(long long i);`

22820 **DESCRIPTION**

22821 Refer to *labs*( ).

22822 **NAME**

22823 lldiv — compute quotient and remainder of a long division

22824 **SYNOPSIS**

22825 `#include <stdlib.h>`

22826 `lldiv_t lldiv(long long` *`numer`*`, long long` *`denom`*`);`

22827 **DESCRIPTION**

22828 Refer to *ldiv*( ).

22829 **NAME**

22830 llrint, llrintf, llrintl, — round to nearest integer value using current rounding direction

22831 **SYNOPSIS**

22832 `#include <math.h>`

22833 `long long llrint(double x);`
22834 `long long llrintf(float x);`
22835 `long long llrintl(long double x);`

22836 **DESCRIPTION**

22837 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22838 conflict between the requirements described here and the ISO C standard is unintentional. This
22839 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22840 These functions shall round their argument to the nearest integer value, rounding according to
22841 the current rounding direction.

22842 An application wishing to check for error situations should set *errno* to zero and call
22843 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
22844 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
22845 zero, an error has occurred.

22846 **RETURN VALUE**

22847 Upon successful completion, these functions shall return the rounded integer value.

22848 MX If *x* is NaN, a domain error shall occur, and an unspecified value is returned.

22849 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

22850 If *x* is –Inf, a domain error shall occur and an unspecified value is returned.

22851 If the correct value is positive and too large to represent as a **long long**, a domain error shall |
22852 occur and an unspecified value is returned. |

22853 If the correct value is negative and too large to represent as a **long long**, a domain error shall |
22854 occur and an unspecified value is returned. |

22855 **ERRORS**

22856 These functions shall fail if:

22857 MX Domain Error — The *x* argument is NaN or ±Inf, or the correct value is not representable as an
22858 integer.

22859 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
22860 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling |
22861 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception |
22862 shall be raised. |

22863 **EXAMPLES**

22864 None.

22865 **APPLICATION USAGE**

22866 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
22867 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

22868 **RATIONALE**

22869 These functions provide floating-to-integer conversions. They round according to the current
22870 rounding direction. If the rounded value is outside the range of the return type, the numeric
22871 result is unspecified and the invalid floating-point exception is raised. When they raise no other
22872 floating-point exception and the result differs from the argument, they raise the inexact

22873 floating-point exception.

22874 **FUTURE DIRECTIONS**

22875 None.

22876 **SEE ALSO**

22877 *feclearexcept*( ), *fetestexcept*( ), *lrint*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section |
22878 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

22879 **CHANGE HISTORY**

22880 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

22881 **NAME**

22882 llround, llroundf, llroundl, — round to nearest integer value

22883 **SYNOPSIS**

22884 `#include <math.h>`

22885 `long long llround(double x);`
22886 `long long llroundf(float x);`
22887 `long long llroundl(long double x);`

22888 **DESCRIPTION**

22889 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22890 conflict between the requirements described here and the ISO C standard is unintentional. This
22891 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22892 These functions shall round their argument to the nearest integer value, rounding halfway cases
22893 away from zero, regardless of the current rounding direction.

22894 An application wishing to check for error situations should set *errno* to zero and call
22895 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
22896 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
22897 zero, an error has occurred.

22898 **RETURN VALUE**

22899 Upon successful completion, these functions shall return the rounded integer value.

22900 MX If *x* is NaN, a domain error shall occur, and an unspecified value is returned.

22901 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

22902 If *x* is –Inf, a domain error shall occur and an unspecified value is returned.

22903 If the correct value is positive and too large to represent as a **long long**, a domain error shall |
22904 occur and an unspecified value is returned.

22905 If the correct value is negative and too large to represent as a **long long**, a domain error shall |
22906 occur and an unspecified value is returned.

22907 **ERRORS**

22908 These functions shall fail if:

22909 MX

Domain Error	The *x* argument is NaN or ±Inf, or the correct value is not representable as an integer.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

22915 **EXAMPLES**

22916 None.

22917 **APPLICATION USAGE**

22918 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
22919 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

22920 **RATIONALE**

22921 These functions provide floating-to-integer conversions. They round according to the current
22922 rounding direction. If the rounded value is outside the range of the return type, the numeric
22923 result is unspecified and the invalid floating-point exception is raised. When they raise no other
22924 floating-point exception and the result differs from the argument, they raise the inexact

22925 floating-point exception.

22926 These functions differ from the *llrint*( ) functions in that the default rounding direction for the
22927 *llround*( ) functions round halfway cases away from zero and need not raise the inexact floating-
22928 point exception for non-integer arguments that round to within the range of the return type.

22929 **FUTURE DIRECTIONS**

22930 None.

22931 **SEE ALSO**

22932 *feclearexcept*( ), *fetestexcept*( ), *lround*( ), the Base Definitions volume of IEEE Std 1003.1-200x, |
22933 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

22934 **CHANGE HISTORY**

22935 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

22936 **NAME**

22937 localeconv — return locale-specific information

22938 **SYNOPSIS**

22939
```
#include <locale.h>
```

22940
```
struct lconv *localeconv(void);
```

22941 **DESCRIPTION**

22942 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22943 conflict between the requirements described here and the ISO C standard is unintentional. This
22944 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22945 The *localeconv*( ) function shall set the components of an object with the type **struct lconv** with
22946 the values appropriate for the formatting of numeric quantities (monetary and otherwise)
22947 according to the rules of the current locale.

22948 The members of the structure with type **char *** are pointers to strings, any of which (except
22949 **decimal_point**) can point to " ", to indicate that the value is not available in the current locale or
22950 is of zero length. The members with type **char** are non-negative numbers, any of which can be
22951 {CHAR_MAX} to indicate that the value is not available in the current locale.

22952 The members include the following:

22953 **char *decimal_point**
22954 The radix character used to format non-monetary quantities.

22955 **char *thousands_sep**
22956 The character used to separate groups of digits before the decimal-point character in
22957 formatted non-monetary quantities.

22958 **char *grouping**
22959 A string whose elements taken as one-byte integer values indicate the size of each group of
22960 digits in formatted non-monetary quantities.

22961 **char *int_curr_symbol**
22962 The international currency symbol applicable to the current locale. The first three
22963 characters contain the alphabetic international currency symbol in accordance with those
22964 specified in the ISO 4217:1995 standard. The fourth character (immediately preceding the
22965 null byte) is the character used to separate the international currency symbol from the
22966 monetary quantity.

22967 **char *currency_symbol**
22968 The local currency symbol applicable to the current locale.

22969 **char *mon_decimal_point**
22970 The radix character used to format monetary quantities.

22971 **char *mon_thousands_sep**
22972 The separator for groups of digits before the decimal-point in formatted monetary
22973 quantities.

22974 **char *mon_grouping**
22975 A string whose elements taken as one-byte integer values indicate the size of each group of
22976 digits in formatted monetary quantities.

22977 **char *positive_sign**
22978 The string used to indicate a non-negative valued formatted monetary quantity.

char *negative_sign
The string used to indicate a negative valued formatted monetary quantity.

char int_frac_digits
The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.

char frac_digits
The number of fractional digits (those after the decimal-point) to be displayed in a formatted monetary quantity.

char p_cs_precedes
Set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a non-negative formatted monetary quantity. Set to 0 if the symbol succeeds the value.

char p_sep_by_space
Set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a non-negative formatted monetary quantity. Set to 1 if a space separates the symbol from the value; XSI and set to 2 if a space separates the symbol and the sign string, if adjacent.

char n_cs_precedes
Set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a negative formatted monetary quantity. Set to 0 if the symbol succeeds the value.

char n_sep_by_space
Set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a negative formatted monetary quantity. Set to 1 if a space separates the symbol from the value; XSI and set to 2 if a space separates the symbol and the sign string, if adjacent.

char p_sign_posn
Set to a value indicating the positioning of the **positive_sign** for a non-negative formatted monetary quantity.

char n_sign_posn
Set to a value indicating the positioning of the **negative_sign** for a negative formatted monetary quantity.

char int_p_cs_precedes
Set to 1 or 0 if the **int_curr_symbol** respectively precedes or succeeds the value for a non-negative internationally formatted monetary quantity. |

char int_n_cs_precedes
Set to 1 or 0 if the **int_curr_symbol** respectively precedes or succeeds the value for a negative internationally formatted monetary quantity. |

char int_p_sep_by_space
Set to a value indicating the separation of the **int_curr_symbol**, the sign string, and the value for a non-negative internationally formatted monetary quantity. |

char int_n_sep_by_space
Set to a value indicating the separation of the **int_curr_symbol**, the sign string, and the value for a negative internationally formatted monetary quantity. |

char int_p_sign_posn
Set to a value indicating the positioning of the **positive_sign** for a non-negative internationally formatted monetary quantity.

char int_n_sign_posn
Set to a value indicating the positioning of the **negative_sign** for a negative internationally

formatted monetary quantity.

The elements of **grouping** and **mon_grouping** are interpreted according to the following:

{CHAR_MAX} No further grouping is to be performed.

0 The previous element is to be repeatedly used for the remainder of the digits.

other The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The values of **p_sep_by_space**, **n_sep_by_space**, **int_p_sep_by_space**, and **int_n_sep_by_space** are interpreted according to the following:

0 No space separates the currency symbol and value.

1 If the currency symbol and sign string are adjacent, a space separates them from the value; otherwise, a space separates the currency symbol from the value.

2 If the currency symbol and sign string are adjacent, a space separates them; otherwise, a space separates the sign string from the value.

For **int_p_sep_by_space** and **int_n_sep_by_space**, the fourth character of **int_curr_symbol** is used instead of a space.

The values of **p_sign_posn**, **n_sign_posn**, **int_p_sign_posn**, and **int_n_sign_posn** are interpreted according to the following:

0 Parentheses surround the quantity and **currency_symbol** or **int_curr_symbol**.

1 The sign string precedes the quantity and **currency_symbol** or **int_curr_symbol**.

2 The sign string succeeds the quantity and **currency_symbol** or **int_curr_symbol**.

3 The sign string immediately precedes the **currency_symbol** or **int_curr_symbol**.

4 The sign string immediately succeeds the **currency_symbol** or **int_curr_symbol**.

The implementation shall behave as if no function in this volume of IEEE Std 1003.1-200x calls *localeconv*().

CX The *localeconv*() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *localeconv*() function shall return a pointer to the filled-in object. The application shall not modify the structure pointed to by the return value which may be overwritten by a subsequent call to *localeconv*(). In addition, calls to *setlocale*() with the categories *LC_ALL*, *LC_MONETARY*, or *LC_NUMERIC* may overwrite the contents of the structure.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The following table illustrates the rules which may be used by four countries to format monetary quantities.

Country	Positive Format	Negative Format	International Format
Italy	L.1.230	–L.1.230	ITL.1.230
Netherlands	F 1.234,56	F –1.234,56	NLG 1.234,56
Norway	kr1.234,56	kr1.234,56–	NOK 1.234,56
Switzerland	SFrs.1,234.56	SFrs.1,234.56C	CHF 1,234.56

For these four countries, the respective values for the monetary members of the structure returned by *localeconv*( ) are:

	Italy	Netherlands	Norway	Switzerland
int_curr_symbol	"ITL."	"NLG "	"NOK "	"CHF "
currency_symbol	"L."	"F"	"kr"	"SFrs."
mon_decimal_point	""	","	","	"."
mon_thousands_sep	"."	"."	"."	","
mon_grouping	"\3"	"\3"	"\3"	"\3"
positive_sign	""	""	""	""
negative_sign	"-"	"-"	"-"	"C"
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_space	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2
int_p_cs_precedes	1	1	1	1
int_n_cs_precedes	1	1	1	1
int_p_sep_by_space	0	0	0	0
int_n_sep_by_space	0	0	0	0
int_p_sign_posn	1	1	1	1
int_n_sign_posn	1	4	4	2

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalpha( ), *isascii*( ), *nl_langinfo*( ), *printf*( ), *scanf*( ), *setlocale*( ), *strcat*( ), *strchr*( ), *strcmp*( ), *strcoll*( ), *strcpy*( ), *strftime*( ), *strlen*( ), *strpbrk*( ), *strspn*( ), *strtok*( ), *strxfrm*( ), *strtod*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<langinfo.h>**, **<locale.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

Issue 6

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

The RETURN VALUE section is rewritten to avoid use of the term ‘‘must’’.

This reference page is updated for alignment with the ISO/IEC 9899:1999 standard. |

ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated. |

23107 **NAME**

23108 localtime, localtime_r — convert a time value to a broken-down local time

23109 **SYNOPSIS**

23110 ```
#include <time.h>
```

23111 ```
struct tm *localtime(const time_t *timer);
```

23112 TSF ```
struct tm *localtime_r(const time_t *restrict timer,
    struct tm *restrict result);
```

23113

23114

23115 **DESCRIPTION**

23116 CX For *localtime*( ): The functionality described on this reference page is aligned with the ISO C
23117 standard. Any conflict between the requirements described here and the ISO C standard is
23118 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23119 The *localtime*( ) function shall convert the time in seconds since the Epoch pointed to by *timer*
23120 into a broken-down time, expressed as a local time. The function corrects for the timezone and
23121 CX any seasonal time adjustments. Local timezone information is used as though *localtime*( ) calls
23122 *tzset*( ).

23123 The relationship between a time in seconds since the Epoch used as an argument to *localtime*( ) |
23124 and the **tm** structure (defined in the **<time.h>** header) is that the result shall be as specified in the |
23125 expression given in the definition of seconds since the Epoch (see the Base Definitions volume of |
23126 IEEE Std 1003.1-200x, Section 4.14, Seconds Since the Epoch) corrected for timezone and any |
23127 seasonal time adjustments, where the names in the structure and in the expression correspond. |

23128 TSF The same relationship shall apply for *localtime_r*( ). |

23129 CX The *localtime*( ) function need not be reentrant. A function that is not required to be reentrant is |
23130 not required to be thread-safe. |

23131 The *asctime*( ), *ctime*( ), *gmtime*( ), and *localtime*( ) functions shall return values in one of two static |
23132 objects: a broken-down time structure and an array of type **char**. Execution of any of the |
23133 functions may overwrite the information returned in either of these objects by any of the other |
23134 functions. |

23135 TSF The *localtime_r*( ) function shall convert the time in seconds since the Epoch pointed to by *timer*
23136 into a broken-down time stored in the structure to which *result* points. The *localtime_r*( ) function
23137 shall also return a pointer to that same structure.

23138 Unlike *localtime*( ), the reentrant version is not required to set *tzname*.

23139 **RETURN VALUE**

23140 The *localtime*( ) function shall return a pointer to the broken-down time structure.

23141 TSF Upon successful completion, *localtime_r*( ) shall return a pointer to the structure pointed to by
23142 the argument *result*.

23143 **ERRORS**

23144 No errors are defined.
```

EXAMPLES

Getting the Local Date and Time

The following example uses the *time*() function to calculate the time elapsed, in seconds, since January 1, 1970 0:00 UTC (the Epoch), *localtime*() to convert that value to a broken-down time, and *asctime*() to convert the broken-down time values into a printable string.

```
#include <stdio.h>
#include <time.h>

main()
{
    time_t result;

    result = time(NULL);
    printf("%s%ld secs since the Epoch\n",
        asctime(localtime(&result)),
            (long)result);
    return(0);
}
```

This example writes the current time to *stdout* in a form like this:

```
Wed Jun 26 10:32:15 1996
835810335 secs since the Epoch
```

Getting the Modification Time for a File

The following example gets the modification time for a file. The *localtime*() function converts the **time_t** value of the last modification date, obtained by a previous call to *stat*(), into a **tm** structure that contains the year, month, day, and so on.

```
#include <time.h>
...
struct stat statbuf;
...
tm = localtime(&statbuf.st_mtime);
...
```

Timing an Event

The following example gets the current time, converts it to a string using *localtime*() and *asctime*(), and prints it to standard output using *fputs*(). It then prints the number of minutes to an event being timed.

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
fputs(asctime(localtime(&now)), stdout);
printf("There are still %d minutes to the event.\n",
```

```
            minutes_to_event);
        ...
```

APPLICATION USAGE

The *localtime_r*( ) function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime( ), *clock*( ), *ctime*( ), *difftime*( ), *getdate*( ), *gmtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A note indicating that the *localtime*( ) function need not be reentrant is added to the DESCRIPTION.

The *localtime_r*( ) function is included for alignment with the POSIX Threads Extension.

Issue 6

The *localtime_r*( ) function is marked as part of the Thread-Safe Functions option.

Extensions beyond the ISO C standard are now marked.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The **restrict** keyword is added to the *localtime_r*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

Examples are added.

NAME

lockf — record locking on files

SYNOPSIS

XSI
```
#include <unistd.h>

int lockf(int fildes, int function, off_t size);
```

DESCRIPTION

The *lockf*( ) function shall lock sections of a file with advisory-mode locks. Calls to *lockf*( ) from other threads which attempt to lock the locked file section shall either return an error value or block until the section becomes unlocked. All the locks for a process are removed when the process terminates. Record locking with *lockf*( ) shall be supported for regular files and may be supported for other files.

The *fildes* argument is an open file descriptor. To establish a lock with this function, the file descriptor shall be opened with write-only permission (O_WRONLY) or with read/write permission (O_RDWR).

The *function* argument is a control value which specifies the action to be taken. The permissible values for *function* are defined in **<unistd.h>** as follows:

Function	Description
F_ULOCK	Unlock locked sections.
F_LOCK	Lock a section for exclusive use.
F_TLOCK	Test and lock a section for exclusive use.
F_TEST	Test a section for locks by other processes.

F_TEST shall detect if a lock by another process is present on the specified section.

F_LOCK and F_TLOCK shall both lock a section of a file if the section is available.

F_ULOCK shall remove locks from a section of the file.

The *size* argument is the number of contiguous bytes to be locked or unlocked. The section to be locked or unlocked starts at the current offset in the file and extends forward for a positive size or backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is 0, the section from the current offset through the largest possible file offset shall be locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file to be locked because locks may exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent locked sections would occur, the sections shall be combined into a single locked section. If the request would cause the number of locks to exceed a system-imposed limit, the request shall fail.

F_LOCK and F_TLOCK requests differ only by the action taken if the section is not available. F_LOCK shall block the calling thread until the section is available. F_TLOCK shall cause the function to fail if the section is already locked by another process.

File locks shall be released on first close by the locking process of any file descriptor for the file.

F_ULOCK requests may release (wholly or in part) one or more locked sections controlled by the process. Locked sections shall be unlocked starting at the current file offset through *size* bytes or to the end-of-file if *size* is (**off_t**)0. When all of a locked section is not released (that is, when the beginning or end of the area to be unlocked falls within a locked section), the remaining portions of that section shall remain locked by the process. Releasing the center portion of a locked

section shall cause the remaining locked beginning and end portions to become two separate locked sections. If the request would cause the number of locks in the system to exceed a system-imposed limit, the request shall fail.

A potential for deadlock occurs if the threads of a process controlling a locked section are blocked by accessing another process' locked section. If the system detects that deadlock would occur, *lockf*( ) shall fail with an [EDEADLK] error.

The interaction between *fcntl*( ) and *lockf*( ) locks is unspecified.

Blocking on a section shall be interrupted by any signal.

An F_ULOCK request in which *size* is non-zero and the offset of the last byte of the requested section is the maximum value for an object of type **off_t**, when the process has an existing lock in which *size* is 0 and which includes the last byte of the requested section, shall be treated as a request to unlock from the start of the requested section with a size equal to 0. Otherwise, an F_ULOCK request shall attempt to unlock only the requested section.

Attempting to lock a section of a file that is associated with a buffered stream produces unspecified results.

RETURN VALUE

Upon successful completion, *lockf*( ) shall return 0. Otherwise, it shall return −1, set *errno* to indicate an error, and existing locks shall not be changed.

ERRORS

The *lockf*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid open file descriptor; or *function* is F_LOCK or F_TLOCK and *fildes* is not a valid file descriptor open for writing.

[EACCES] or [EAGAIN]
The *function* argument is F_TLOCK or F_TEST and the section is already locked by another process.

[EDEADLK] The *function* argument is F_LOCK and a deadlock is detected.

[EINTR] A signal was caught during execution of the function.

[EINVAL] The *function* argument is not one of F_LOCK, F_TLOCK, F_TEST, or F_ULOCK; or *size* plus the current file offset is less than 0.

[EOVERFLOW] The offset of the first, or if *size* is not 0 then the last, byte in the requested section cannot be represented correctly in an object of type **off_t**.

The *lockf*( ) function may fail if:

[EAGAIN] The *function* argument is F_LOCK or F_TLOCK and the file is mapped with *mmap*( ).

[EDEADLK] or [ENOLCK]
The *function* argument is F_LOCK, F_TLOCK, or F_ULOCK, and the request would cause the number of locks to exceed a system-imposed limit.

[EOPNOTSUPP] or [EINVAL]
The implementation does not support the locking of files of the type indicated by the *fildes* argument.

EXAMPLES

Locking a Portion of a File

In the following example, a file named **/home/cnd/mod1** is being modified. Other processes that use locking are prevented from changing it during this process. Only the first 10,000 bytes are locked, and the lock call fails if another process has any part of this area locked already.

```
#include <fcntl.h>
#include <unistd.h>

int fildes;
int status;
...
fildes = open("/home/cnd/mod1", O_RDWR);
status = lockf(fildes, F_TLOCK, (off_t)10000);
```

APPLICATION USAGE

Record-locking should not be used in combination with the *fopen*(), *fread*(), *fwrite*(), and other *stdio* functions. Instead, the more primitive, non-buffered functions (such as *open*()) should be used. Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The *stdio* functions are the most common source of unexpected buffering.

The *alarm*() function may be used to provide a timeout facility in applications requiring it.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm(), *chmod*(), *close*(), *creat*(), *fcntl*(), *fopen*(), *mmap*(), *open*(), *read*(), *write*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Large File Summit extensions are added. In particular, the description of [EINVAL] is clarified and moved from optional to mandatory status.

A note is added to the DESCRIPTION indicating the effects of attempting to lock a section of a file that is associated with a buffered stream.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

23335 **NAME**

23336 log, logf, logl — natural logarithm function

23337 **SYNOPSIS**

23338
```
#include <math.h>
```

23339
```
double log(double x);
```
23340
```
float logf(float x);
```
23341
```
long double logl(long double x);
```

23342 **DESCRIPTION**

23343 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23344 conflict between the requirements described here and the ISO C standard is unintentional. This
23345 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23346 These functions shall compute the natural logarithm of their argument x, $\log_e(x)$.

23347 An application wishing to check for error situations should set *errno* to zero and call
23348 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
23349 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
23350 zero, an error has occurred.

23351 **RETURN VALUE**

23352 Upon successful completion, these functions shall return the natural logarithm of x.

23353 If x is ±0, a pole error shall occur and *log*(), *logf*(), and *logl*() shall return –HUGE_VAL,
23354 –HUGE_VALF, and –HUGE_VALL, respectively.

23355 MX For finite values of x that are less than 0, or if x is –Inf, a domain error shall occur, and either a
23356 NaN (if supported), or an implementation-defined value shall be returned.

23357 MX If x is NaN, a NaN shall be returned.

23358 If x is 1, +0 shall be returned.

23359 If x is +Inf, x shall be returned.

23360 **ERRORS**

23361 These functions shall fail if:

23362 MX Domain Error — The finite value of x is negative, or x is –Inf.

23363 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
23364 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
23365 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
23366 shall be raised.

23367 Pole Error — The value of x is zero.

23368 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
23369 then *errno* shall be set to [ERANGE]. If the integer expression
23370 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-
23371 zero floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exp( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *log10*( ), *log1p*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *logf*( ) and *logl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

23397 **NAME**

23398 log10, log10f, log10l — base 10 logarithm function

23399 **SYNOPSIS**

23400
```
#include <math.h>
```

23401
```
double log10(double x);
```
23402
```
float log10f(float x);
```
23403
```
long double log10l(long double x);
```

23404 **DESCRIPTION**

23405 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23406 conflict between the requirements described here and the ISO C standard is unintentional. This
23407 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23408 These functions shall compute the base 10 logarithm of their argument x, $\log_{10}(x)$.

23409 An application wishing to check for error situations should set *errno* to zero and call
23410 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
23411 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
23412 zero, an error has occurred.

23413 **RETURN VALUE**

23414 Upon successful completion, these functions shall return the base 10 logarithm of x.

23415 If x is ±0, a pole error shall occur and *log10*( ), *log10f*( ), and *log10l*( ) shall return –HUGE_VAL,
23416 –HUGE_VALF, and –HUGE_VALL, respectively.

23417 MX For finite values of x that are less than 0, or if x is –Inf, a domain error shall occur, and either a
23418 NaN (if supported), or an implementation-defined value shall be returned.

23419 MX If x is NaN, a NaN shall be returned.

23420 If x is 1, +0 shall be returned.

23421 If x is +Inf, +Inf shall be returned.

23422 **ERRORS**

23423 These functions shall fail if:

23424 MX Domain Error — The finite value of x is negative, or x is –Inf.

23425 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
23426 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
23427 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
23428 shall be raised.

23429 Pole Error — The value of x is zero.

23430 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
23431 then *errno* shall be set to [ERANGE]. If the integer expression
23432 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-
23433 zero floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *isnan*( ), *log*( ), *pow*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *log10f*( ) and *log10l*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

log1p, log1pf, log1pl — compute a natural logarithm

SYNOPSIS

```
#include <math.h>

double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute $\log_e(1.0 + x)$.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the natural logarithm of 1.0 + *x*.

If *x* is −1, a pole error shall occur and *log1p*( ), *log1pf*( ), and *log1pl*( ) shall return −HUGE_VAL, −HUGE_VALF, and −HUGE_VALL, respectively.

MX For finite values of *x* that are less than −1, or if *x* is −Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, or +Inf, *x* shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

These functions shall fail if:

MX Domain Error — The finite value of *x* is less than −1, or *x* is −Inf.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

Pole Error — The value of *x* is −1.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised.

These functions may fail if:

MX Range Error — The value of *x* is subnormal.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *log*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *log1p*( ) function is no longer marked as an extension.

The *log1pf*( ) and *log1pl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

23528 **NAME**

23529 log2, log2f, log2l — compute base 2 logarithm functions

23530 **SYNOPSIS**

23531
```
#include <math.h>
```

23532
```
double log2(double x);
```
23533
```
float log2f(float x);
```
23534
```
long double log2l(long double x);
```

23535 **DESCRIPTION**

23536 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23537 conflict between the requirements described here and the ISO C standard is unintentional. This
23538 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23539 These functions shall compute the base 2 logarithm of their argument x, $\log_2(x)$. |

23540 An application wishing to check for error situations should set *errno* to zero and call
23541 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
23542 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
23543 zero, an error has occurred.

23544 **RETURN VALUE**

23545 Upon successful completion, these functions shall return the base 2 logarithm of x.

23546 If x is ±0, a pole error shall occur and *log2*(), *log2f*(), and *log2l*() shall return –HUGE_VAL,
23547 –HUGE_VALF, and –HUGE_VALL, respectively.

23548 MX For finite values of x that are less than 0, or if x is –Inf a domain error shall occur, and either a
23549 NaN (if supported), or an implementation-defined value shall be returned.

23550 MX If x is NaN, a NaN shall be returned.

23551 If x is 1, +0 shall be returned.

23552 If x is +Inf, x shall be returned.

23553 **ERRORS**

23554 These functions shall fail if:

23555 MX Domain Error — The finite value of x is less than zero, or x is –Inf.

23556 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
23557 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling |
23558 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception |
23559 shall be raised. |

23560 Pole Error — The value of x is zero.

23561 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
23562 then *errno* shall be set to [ERANGE]. If the integer expression |
23563 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by- |
23564 zero floating-point exception shall be raised. |

23565 **EXAMPLES**

23566 None.

23567 **APPLICATION USAGE**

23568 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
23569 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23570 **RATIONALE**

23571 None.

23572 **FUTURE DIRECTIONS**

23573 None.

23574 **SEE ALSO**

23575 *feclearexcept*( ), *fetestexcept*( ), *log*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section |
23576 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

23577 **CHANGE HISTORY**

23578 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

23579 **NAME**

23580 logb, logbf, logbl — radix-independent exponent

23581 **SYNOPSIS**

23582 `#include <math.h>`

23583 `double logb(double x);`
23584 `float logbf(float x);`
23585 `long double logbl(long double x);`

23586 **DESCRIPTION**

23587 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23588 conflict between the requirements described here and the ISO C standard is unintentional. This
23589 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23590 These functions shall compute the exponent of *x*, which is the integral part of $\log_r |x|$, as a
23591 signed floating-point value, for non-zero *x*, where *r* is the radix of the machine's floating-point
23592 arithmetic, which is the value of FLT_RADIX defined in the **<float.h>** header.

23593 If *x* is subnormal it is treated a though it were normalized; thus for finite positive *x*:

23594 $1 <= x * FLT\_RADIX^{-logb(x)} < FLT\_RADIX$

23595 An application wishing to check for error situations should set *errno* to zero and call
23596 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
23597 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
23598 zero, an error has occurred.

23599 **RETURN VALUE**

23600 Upon successful completion, these functions shall return the exponent of *x*.

23601 If *x* is ±0, a pole error shall occur and *logb*(), *logbf*(), and *logbl*() shall return –HUGE_VAL,
23602 –HUGE_VALF, and –HUGE_VALL, respectively.

23603 MX If *x* is NaN, a NaN shall be returned.

23604 If *x* is ±Inf, +Inf shall be returned.

23605 **ERRORS**

23606 These functions shall fail if:

23607 Pole Error The value of *x* is ±0.

23608 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
23609 then *errno* shall be set to [ERANGE]. If the integer expression |
23610 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by- |
23611 zero floating-point exception shall be raised. |

23612 **EXAMPLES**

23613 None.

23614 **APPLICATION USAGE**

23615 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
23616 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23617 **RATIONALE**

23618 None.

23619 **FUTURE DIRECTIONS**

23620 None.

23621 **SEE ALSO**

23622 *feclearexcept*( ), *fetestexcept*( ), *ilogb*( ), *scalb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, |
23623 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<float.h>**, **<math.h>** |

23624 **CHANGE HISTORY**

23625 First released in Issue 4, Version 2.

23626 **Issue 5**

23627 Moved from X/OPEN UNIX extension to BASE.

23628 **Issue 6**

23629 The *logb*( ) function is no longer marked as an extension.

23630 The *logbf*( ) and *logbl*( ) functions are added for alignment with the ISO/IEC 9899: 1999 standard.

23631 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
23632 revised to align with the ISO/IEC 9899: 1999 standard.

23633 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are
23634 marked.

23635 **NAME**

23636 logf — natural logarithm function

23637 **SYNOPSIS**

23638 `#include <math.h>`

23639 `float logf(float x);`

23640 **DESCRIPTION**

23641 Refer to *log*( ).

23642 **NAME**

23643 logl — natural logarithm function

23644 **SYNOPSIS**

23645 `#include <math.h>`

23646 `long double logl(long double x);`

23647 **DESCRIPTION**

23648 Refer to *log*( ).

23649 **NAME**

23650 longjmp — non-local goto

23651 **SYNOPSIS**

23652
```
#include <setjmp.h>
```

23653
```
void longjmp(jmp_buf env, int val);
```

23654 **DESCRIPTION**

23655 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23656 conflict between the requirements described here and the ISO C standard is unintentional. This
23657 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23658 The *longjmp*( ) function shall restore the environment saved by the most recent invocation of
23659 *setjmp*( ) in the same thread, with the corresponding **jmp_buf** argument. If there is no such
23660 invocation, or if the function containing the invocation of *setjmp*( ) has terminated execution in
23661 the interim, or if the invocation of *setjmp*( ) was within the scope of an identifier with variably
23662 CX modified type and execution has left that scope in the interim, the behavior is undefined. It is
23663 unspecified whether *longjmp*( ) restores the signal mask, leaves the signal mask unchanged, or
23664 restores it to its value at the time *setjmp*( ) was called.

23665 All accessible objects have values, and all other components of the abstract machine have state
23666 (for example, floating-point status flags and open files), as of the time *longjmp*( ) was called,
23667 except that the values of objects of automatic storage duration are unspecified if they meet all
23668 the following conditions:

23669 • They are local to the function containing the corresponding *setjmp*( ) invocation.

23670 • They do not have volatile-qualified type.

23671 • They are changed between the *setjmp*( ) invocation and *longjmp*( ) call.

23672 CX As it bypasses the usual function call and return mechanisms, *longjmp*( ) shall execute correctly
23673 in contexts of interrupts, signals, and any of their associated functions. However, if *longjmp*( ) is
23674 invoked from a nested signal handler (that is, from a function invoked as a result of a signal
23675 raised during the handling of another signal), the behavior is undefined.

23676 The effect of a call to *longjmp*( ) where initialization of the **jmp_buf** structure was not performed
23677 in the calling thread is undefined.

23678 **RETURN VALUE**

23679 After *longjmp*( ) is completed, program execution continues as if the corresponding invocation of
23680 *setjmp*( ) had just returned the value specified by *val*. The *longjmp*( ) function shall not cause
23681 *setjmp*( ) to return 0; if *val* is 0, *setjmp*( ) shall return 1.

23682 **ERRORS**

23683 No errors are defined.

23684 **EXAMPLES**

23685 None.

23686 **APPLICATION USAGE**

23687 Applications whose behavior depends on the value of the signal mask should not use *longjmp*( )
23688 and *setjmp*( ), since their effect on the signal mask is unspecified, but should instead use the
23689 *siglongjmp*( ) and *sigsetjmp*( ) functions (which can save and restore the signal mask under
23690 application control).

23691 **RATIONALE**

23692 None.

23693 **FUTURE DIRECTIONS**

23694 None.

23695 **SEE ALSO**

23696 *setjmp*( ), *sigaction*( ), *siglongjmp*( ), *sigsetjmp*( ), the Base Definitions volume of
23697 IEEE Std 1003.1-200x, **<setjmp.h>**

23698 **CHANGE HISTORY**

23699 First released in Issue 1. Derived from Issue 1 of the SVID.

23700 **Issue 5**

23701 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

23702 **Issue 6**

23703 Extensions beyond the ISO C standard are now marked.

23704 The following new requirements on POSIX implementations derive from alignment with the
23705 Single UNIX Specification:

23706 - The DESCRIPTION now explicitly makes *longjmp*( )'s effect on the signal mask unspecified.

23707 The DESCRIPTION is updated for alignment with the ISO / IEC 9899: 1999 standard.

23708 **NAME**

23709 lrand48 — generate uniformly distributed pseudo-random non-negative long integers

23710 **SYNOPSIS**

23711 XSI
```
#include <stdlib.h>

long lrand48(void);
```
23712

23713

23714 **DESCRIPTION**

23715 Refer to *drand48*( ).

23716 **NAME**

23717 lrint, lrintf, lrintl — round to nearest integer value using current rounding direction

23718 **SYNOPSIS**

23719
```
#include <math.h>
```

23720
```
long lrint(double x);
```
23721
```
long lrintf(float x);
```
23722
```
long lrintl(long double x);
```

23723 **DESCRIPTION**

23724 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23725 conflict between the requirements described here and the ISO C standard is unintentional. This
23726 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23727 These functions shall round their argument to the nearest integer value, rounding according to
23728 the current rounding direction.

23729 An application wishing to check for error situations should set *errno* to zero and call
23730 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
23731 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
23732 zero, an error has occurred.

23733 **RETURN VALUE**

23734 Upon successful completion, these functions shall return the rounded integer value.

23735 MX If *x* is NaN, a domain error shall occur, and an unspecified value is returned.

23736 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

23737 If *x* is –Inf, a domain error shall occur and an unspecified value is returned.

23738 If the correct value is positive and too large to represent as a **long**, a domain error shall occur |
23739 and an unspecified value is returned.

23740 If the correct value is negative and too large to represent as a **long**, a domain error shall occur |
23741 and an unspecified value is returned.

23742 **ERRORS**

23743 These functions shall fail if:

23744 MX Domain Error The *x* argument is NaN or ±Inf, or the correct value is not representable as an
23745 integer.

23746 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
23747 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling |
23748 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception |
23749 shall be raised. |

23750 **EXAMPLES**

23751 None.

23752 **APPLICATION USAGE**

23753 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
23754 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23755 **RATIONALE**

23756 These functions provide floating-to-integer conversions. They round according to the current
23757 rounding direction. If the rounded value is outside the range of the return type, the numeric
23758 result is unspecified and the invalid floating-point exception is raised. When they raise no other
23759 floating-point exception and the result differs from the argument, they raise the inexact

23760 floating-point exception.

23761 **FUTURE DIRECTIONS**

23762 None.

23763 **SEE ALSO**

23764 *feclearexcept*( ), *fetestexcept*( ), *llrint*( ), the Base Definitions volume of IEEE Std 1003.1-200x, |
23765 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

23766 **CHANGE HISTORY**

23767 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

23768 **NAME**

23769 lround, lroundf, lroundl — round to nearest integer value

23770 **SYNOPSIS**

23771
```
#include <math.h>

long lround(double x);
long lroundf(float x);
long lroundl(long double x);
```

23775 **DESCRIPTION**

23776 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23777 conflict between the requirements described here and the ISO C standard is unintentional. This
23778 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23779 These functions shall round their argument to the nearest integer value, rounding halfway cases
23780 away from zero, regardless of the current rounding direction.

23781 An application wishing to check for error situations should set *errno* to zero and call
23782 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
23783 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
23784 zero, an error has occurred.

23785 **RETURN VALUE**

23786 Upon successful completion, these functions shall return the rounded integer value.

23787 MX If *x* is NaN, a domain error shall occur, and an unspecified value is returned.

23788 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

23789 If *x* is –Inf, a domain error shall occur and an unspecified value is returned.

23790 If the correct value is positive and too large to represent as a **long**, a domain error shall occur |
23791 and an unspecified value is returned.

23792 If the correct value is negative and too large to represent as a **long**, a domain error shall occur |
23793 and an unspecified value is returned.

23794 **ERRORS**

23795 These functions shall fail if:

23796 MX Domain Error The *x* argument is NaN or ±Inf, or the correct value is not representable as an
23797 integer.

23798 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
23799 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling |
23800 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception |
23801 shall be raised. |

23802 **EXAMPLES**

23803 None.

23804 **APPLICATION USAGE**

23805 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
23806 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23807 **RATIONALE**

23808 These functions provide floating-to-integer conversions. They round according to the current
23809 rounding direction. If the rounded value is outside the range of the return type, the numeric
23810 result is unspecified and the invalid floating-point exception is raised. When they raise no other
23811 floating-point exception and the result differs from the argument, they raise the inexact

23812 floating-point exception.

23813 These functions differ from the *lrint*() functions in the default rounding direction, with the
23814 *lround*() functions rounding halfway cases away from zero and needing not to raise the inexact
23815 floating-point exception for non-integer arguments that round to within the range of the return
23816 type.

23817 **FUTURE DIRECTIONS**

23818 None.

23819 **SEE ALSO**

23820 *feclearexcept*(), *fetestexcept*(), *llround*(), the Base Definitions volume of IEEE Std 1003.1-200x, |
23821 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

23822 **CHANGE HISTORY**

23823 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

lsearch, lfind — linear search and update

SYNOPSIS

XSI

```
#include <search.h>

void *lsearch(const void *key, void *base, size_t *nelp, size_t width,
    int (*compar)(const void *, const void *));
void *lfind(const void *key, const void *base, size_t *nelp,
    size_t width, int (*compar)(const void *, const void *));
```

DESCRIPTION

The *lsearch*( ) function shall linearly search the table and return a pointer into the table for the matching entry. If the entry does not occur, it shall be added at the end of the table. The *key* argument points to the entry to be sought in the table. The *base* argument points to the first element in the table. The *width* argument is the size of an element in bytes. The *nelp* argument points to an integer containing the current number of elements in the table. The integer to which *nelp* points shall be incremented if the entry is added to the table. The *compar* argument points to a comparison function which the application shall supply (for example, *strcmp*( )). It is called with two arguments that point to the elements being compared. The application shall ensure that the function returns 0 if the elements are equal, and non-zero otherwise.

The *lfind*( ) function shall be equivalent to *lsearch*( ), except that if the entry is not found, it is not added to the table. Instead, a null pointer is returned.

RETURN VALUE

If the searched for entry is found, both *lsearch*( ) and *lfind*( ) shall return a pointer to it. Otherwise, *lfind*( ) shall return a null pointer and *lsearch*( ) shall return a pointer to the newly added element.

Both functions shall return a null pointer in case of error.

ERRORS

No errors are defined.

EXAMPLES

Storing Strings in a Table

This fragment reads in less than or equal to TABSIZE strings of length less than or equal to ELSIZE and stores them in a table, eliminating duplicates.

```
#include <stdio.h>
#include <string.h>
#include <search.h>

#define TABSIZE 50
#define ELSIZE 120

...
    char line[ELSIZE], tab[TABSIZE][ELSIZE];
    size_t nel = 0;
    ...
    while (fgets(line, ELSIZE, stdin) != NULL && nel < TABSIZE)
        (void) lsearch(line, tab, &nel,
            ELSIZE, (int (*)(const void *, const void *)) strcmp);
    ...
```

Finding a Matching Entry

The following example finds any line that reads "This is a test.".

```
#include <search.h>
#include <string.h>
...
char line[ELSIZE], tab[TABSIZE][ELSIZE];
size_t nel = 0;
char *findline;
void *entry;

findline = "This is a test.\n";

entry = lfind(findline, tab, &nel, ELSIZE, (
    int (*)(const void *, const void *)) strcmp);
```

APPLICATION USAGE

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Undefined results can occur if there is not enough room in the table to add a new item.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

hcreate( ), *tsearch*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<search.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

23894 **NAME**

23895 lseek — move the read/write file offset

23896 **SYNOPSIS**

23897
```
#include <unistd.h>
```

23898
```
off_t lseek(int fildes, off_t offset, int whence);
```

23899 **DESCRIPTION**

23900 The *lseek*( ) function shall set the file offset for the open file description associated with the file
23901 descriptor *fildes*, as follows:

23902 • If *whence* is SEEK_SET, the file offset shall be set to *offset* bytes.

23903 • If *whence* is SEEK_CUR, the file offset shall be set to its current location plus *offset*.

23904 • If *whence* is SEEK_END, the file offset shall be set to the size of the file plus *offset*.

23905 The symbolic constants SEEK_SET, SEEK_CUR, and SEEK_END are defined in **<unistd.h>**.

23906 The behavior of *lseek*( ) on devices which are incapable of seeking is implementation-defined.
23907 The value of the file offset associated with such a device is undefined.

23908 The *lseek*( ) function shall allow the file offset to be set beyond the end of the existing data in the
23909 file. If data is later written at this point, subsequent reads of data in the gap shall return bytes
23910 with the value 0 until data is actually written into the gap.

23911 The *lseek*( ) function shall not, by itself, extend the size of a file.

23912 SHM If *fildes* refers to a shared memory object, the result of the *lseek*( ) function is unspecified.

23913 TYM If *fildes* refers to a typed memory object, the result of the *lseek*( ) function is unspecified.

23914 **RETURN VALUE**

23915 Upon successful completion, the resulting offset, as measured in bytes from the beginning of the
23916 file, shall be returned. Otherwise, (**off_t**)–1 shall be returned, *errno* shall be set to indicate the
23917 error, and the file offset shall remain unchanged.

23918 **ERRORS**

23919 The *lseek*( ) function shall fail if:

23920 [EBADF] The *fildes* argument is not an open file descriptor.

23921 [EINVAL] The *whence* argument is not a proper value, or the resulting file offset would
23922 be negative for a regular file, block special file, or directory.

23923 [EOVERFLOW] The resulting file offset would be a value which cannot be represented
23924 correctly in an object of type **off_t**.

23925 [ESPIPE] The *fildes* argument is associated with a pipe, FIFO, or socket.

23926 **EXAMPLES**

23927 None.

23928 **APPLICATION USAGE**

23929 None.

23930 **RATIONALE**

23931 The ISO C standard includes the functions *fgetpos*( ) and *fsetpos*( ), which work on very large files
23932 by use of a special positioning type.

23933 Although *lseek*( ) may position the file offset beyond the end of the file, this function does not
23934 itself extend the size of the file. While the only function in IEEE Std 1003.1-200x that may directly

extend the size of the file is *write*( ), *truncate*( ), and *ftruncate*( ), several functions originally derived from the ISO C standard, such as *fwrite*( ), *fprintf*( ), and so on, may do so (by causing calls on *write*( )).

An invalid file offset that would cause [EINVAL] to be returned may be both implementation-defined and device-dependent (for example, memory may have few invalid values). A negative file offset may be valid for some devices in some implementations.

The POSIX.1-1990 standard did not specifically prohibit *lseek*( ) from returning a negative offset. Therefore, an application was required to clear *errno* prior to the call and check *errno* upon return to determine whether a return value of (**off_t**)–1 is a negative offset or an indication of an error condition. The standard developers did not wish to require this action on the part of a conforming application, and chose to require that *errno* be set to [EINVAL] when the resulting file offset would be negative for a regular file, block special file, or directory.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [EOVERFLOW] error condition is added. This change is to support large files.

An additional [ESPIPE] error condition is added for sockets.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that *lseek*( ) results are unspecified for typed memory objects.

23967 **NAME**

23968 lstat — get symbolic link status

23969 **SYNOPSIS**

23970 `#include <sys/stat.h>`

23971 `int lstat(const char *restrict path, struct stat *restrict buf);`

23972 **DESCRIPTION**

23973 The *lstat*( ) function shall be equivalent to *stat*( ), except when *path* refers to a symbolic link. In |
23974 that case *lstat*( ) shall return information about the link, while *stat*( ) shall return information
23975 about the file the link references.

23976 For symbolic links, the *st_mode* member shall contain meaningful information when used with
23977 the file type macros, and the *st_size* member shall contain the length of the pathname contained |
23978 in the symbolic link. File mode bits and the contents of the remaining members of the **stat** |
23979 structure are unspecified. The value returned in the *st_size* member is the length of the contents
23980 of the symbolic link, and does not count any trailing null.

23981 **RETURN VALUE**

23982 Upon successful completion, *lstat*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to
23983 indicate the error.

23984 **ERRORS**

23985 The *lstat*( ) function shall fail if:

23986 [EACCES] A component of the path prefix denies search permission.

23987 [EIO] An error occurred while reading from the file system.

23988 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
23989 argument.

23990 [ENAMETOOLONG]
23991 The length of a pathname exceeds {PATH_MAX} or a pathname component is |
23992 longer than {NAME_MAX}. |

23993 [ENOTDIR] A component of the path prefix is not a directory.

23994 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

23995 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file
23996 serial number cannot be represented correctly in the structure pointed to by
23997 *buf*.

23998 The *lstat*( ) function may fail if:

23999 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
24000 resolution of the *path* argument.

24001 [ENAMETOOLONG]
24002 As a result of encountering a symbolic link in resolution of the *path* argument, |
24003 the length of the substituted pathname string exceeded {PATH_MAX}. |

24004 [EOVERFLOW] One of the members is too large to store into the structure pointed to by the
24005 *buf* argument.

EXAMPLES

Obtaining Symbolic Link Status Information

The following example shows how to obtain status information for a symbolic link named **/modules/pass1**. The structure variable *buffer* is defined for the **stat** structure. If the *path* argument specified the filename for the file pointed to by the symbolic link (**/home/cnd/mod1**), the results of calling the function would be the same as those returned by a call to the *stat*( ) function.

```
#include <sys/stat.h>

struct stat buffer;
int status;
...
status = lstat("/modules/pass1", &buffer);
```

APPLICATION USAGE

None.

RATIONALE

The *lstat*( ) function is not required to update the time-related fields if the named file is not a symbolic link. While the *st_uid*, *st_gid*, *st_atime*, *st_mtime*, and *st_ctime* members of the **stat** structure may apply to a symbolic link, they are not required to do so. No functions in IEEE Std 1003.1-200x are required to maintain any of these time fields.

FUTURE DIRECTIONS

None.

SEE ALSO

fstat( ), *readlink*( ), *stat*( ), *symlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Large File Summit extensions are added.

Issue 6

The following changes were made to align with the IEEE P1003.1a draft standard:

- This function is now mandatory.
- The [ELOOP] optional error condition is added.

The **restrict** keyword is added to the *lstat*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

NAME

makecontext, swapcontext — manipulate user contexts

SYNOPSIS

XSI

```
#include <ucontext.h>

void makecontext(ucontext_t *ucp, void (*func)(void),
    int argc, ...);
int swapcontext(ucontext_t *restrict oucp,
    const ucontext_t *restrict ucp);
```

DESCRIPTION

The *makecontext*( ) function shall modify the context specified by *ucp*, which has been initialized using *getcontext*( ). When this context is resumed using *swapcontext*( ) or *setcontext*( ), program execution shall continue by calling *func*, passing it the arguments that follow *argc* in the *makecontext*( ) call.

Before a call is made to *makecontext*( ), the application shall ensure that the context being modified has a stack allocated for it. The application shall ensure that the value of *argc* matches the number of integer arguments passed to *func*; otherwise, the behavior is undefined.

The *uc_link* member is used to determine the context that shall be resumed when the context being modified by *makecontext*( ) returns. The application shall ensure that the *uc_link* member is initialized prior to the call to *makecontext*( ).

The *swapcontext*( ) function shall save the current context in the context structure pointed to by *oucp* and shall set the context to the context structure pointed to by *ucp*.

RETURN VALUE

Upon successful completion, *swapcontext*( ) shall return 0. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *swapcontext*( ) function shall fail if:

[ENOMEM] The *ucp* argument does not have enough stack left to complete the operation.

EXAMPLES

The following example illustrates the use of *makecontext*( ):

```
#include <stdio.h>
#include <ucontext.h>

static ucontext_t ctx[3];

static void
f1 (void)
{
    puts("start f1");
    swapcontext(&ctx[1], &ctx[2]);
    puts("finish f1");
}

static void
f2 (void)
{
    puts("start f2");
    swapcontext(&ctx[2], &ctx[1]);
```

```
24086           puts("finish f2");                                          |
24087       }                                                               |
24088       int                                                             |
24089       main (void)                                                     |
24090       {                                                               |
24091           char st1[8192];                                             |
24092           char st2[8192];                                             |
24093           getcontext(&ctx[1]);                                        |
24094           ctx[1].uc_stack.ss_sp = st1;                                |
24095           ctx[1].uc_stack.ss_size = sizeof st1;                       |
24096           ctx[1].uc_link = &ctx[0];                                   |
24097           makecontext(&ctx[1], f1, 0);                                |
24098           getcontext(&ctx[2]);                                        |
24099           ctx[2].uc_stack.ss_sp = st2;                                |
24100           ctx[2].uc_stack.ss_size = sizeof st2;                       |
24101           ctx[2].uc_link = &ctx[1];                                   |
24102           makecontext(&ctx[2], f2, 0);                                |
24103           swapcontext(&ctx[0], &ctx[2]);                              |
24104           return 0;                                                   |
24105       }                                                               |
```

24106 **APPLICATION USAGE** |

24107 None.

24108 **RATIONALE**

24109 None.

24110 **FUTURE DIRECTIONS**

24111 None.

24112 **SEE ALSO**

24113 *exit*( ), *getcontext*( ), *sigaction*( ), *sigprocmask*( ), the Base Definitions volume of
24114 IEEE Std 1003.1-200x, **<ucontext.h>**

24115 **CHANGE HISTORY**

24116 First released in Issue 4, Version 2.

24117 **Issue 5**

24118 Moved from X/OPEN UNIX extension to BASE.

24119 In the ERRORS section, the description of [ENOMEM] is changed to apply to *swapcontext*( ) only.

24120 **Issue 6**

24121 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

24122 The **restrict** keyword is added to the *swapcontext*( ) prototype for alignment with the
24123 ISO/IEC 9899: 1999 standard.

NAME

malloc — a memory allocator

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *malloc*( ) function shall allocate unused space for an object whose size in bytes is specified by *size* and whose value is unspecified.

The order and contiguity of storage allocated by successive calls to *malloc*( ) is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned. If the size of the space requested is 0, the behavior is implementation-defined: the value returned shall be either a null pointer or a unique pointer.

RETURN VALUE

Upon successful completion with *size* not equal to 0, *malloc*( ) shall return a pointer to the allocated space. If *size* is 0, either a null pointer or a unique pointer that can be successfully passed to *free*( ) shall be returned. Otherwise, it shall return a null pointer CX and set *errno* to indicate the error.

ERRORS

The *malloc*( ) function shall fail if:

CX [ENOMEM] Insufficient storage space is available.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

calloc( ), *free*( ), *realloc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the requirement to set *errno* to indicate an error is added.
- The [ENOMEM] error condition is added.

24169 **NAME**

24170 mblen — get number of bytes in a character

24171 **SYNOPSIS**

24172 `#include <stdlib.h>`

24173 `int mblen(const char *s, size_t n);`

24174 **DESCRIPTION**

24175 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24176 conflict between the requirements described here and the ISO C standard is unintentional. This
24177 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24178 If *s* is not a null pointer, *mblen*( ) shall determine the number of bytes constituting the character |
24179 pointed to by *s*. Except that the shift state of *mbtowc*( ) is not affected, it shall be equivalent to: |

24180 `mbtowc((wchar_t *)0, s, n);`

24181 The implementation shall behave as if no function defined in this volume of
24182 IEEE Std 1003.1-200x calls *mblen*( ).

24183 The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a
24184 state-dependent encoding, this function shall be placed into its initial state by a call for which its |
24185 character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null |
24186 pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a |
24187 null pointer shall cause this function to return a non-zero value if encodings have state |
24188 dependency, and 0 otherwise. If the implementation employs special bytes to change the shift |
24189 state, these bytes shall not produce separate wide-character codes, but shall be grouped with an |
24190 adjacent character. Changing the *LC_CTYPE* category causes the shift state of this function to be |
24191 unspecified. |

24192 **RETURN VALUE**

24193 If *s* is a null pointer, *mblen*( ) shall return a non-zero or 0 value, if character encodings,
24194 respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *mblen*( ) shall
24195 either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the
24196 character (if the next *n* or fewer bytes form a valid character), or return –1 (if they do not form a
24197 CX valid character) and may set *errno* to indicate the error. In no case shall the value returned be
24198 greater than *n* or the value of the {MB_CUR_MAX} macro.

24199 **ERRORS**

24200 The *mblen*( ) function may fail if:

24201 XSI [EILSEQ] Invalid character sequence is detected.

24202 **EXAMPLES**

24203 None.

24204 **APPLICATION USAGE**

24205 None.

24206 **RATIONALE**

24207 None.

24208 **FUTURE DIRECTIONS**

24209 None.

24210 **SEE ALSO**

24211 *mbtowc*( ), *mbstowcs*( ), *wctomb*( ), *wcstombs*( ), the Base Definitions volume of
24212 IEEE Std 1003.1-200x, **<stdlib.h>**

24213 **CHANGE HISTORY**

24214 First released in Issue 4. Aligned with the ISO C standard.

NAME

mbrlen — get number of bytes in a character (restartable)

SYNOPSIS

```
#include <wchar.h>

size_t mbrlen(const char *restrict s, size_t n,
    mbstate_t *restrict ps);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

If *s* is not a null pointer, *mbrlen*( ) shall determine the number of bytes constituting the character pointed to by *s*. It shall be equivalent to:

```
mbstate_t internal;
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal);
```

If *ps* is a null pointer, the *mbrlen*( ) function shall use its own internal **mbstate_t** object, which is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* shall be used to completely describe the current conversion state of the associated character sequence. The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *mbrlen*( ).

The behavior of this function is affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

The *mbrlen*( ) function shall return the first of the following that applies:

0 — If the next *n* or fewer bytes complete the character that corresponds to the null wide character.

positive — If the next *n* or fewer bytes complete a valid character; the value returned shall be the number of bytes that complete the character.

(**size_t**)–2 — If the next *n* bytes contribute to an incomplete but potentially valid character, and all *n* bytes have been processed. When *n* has at least the value of the {MB_CUR_MAX} macro, this case can only occur if *s* points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

(**size_t**)–1 — If an encoding error occurs, in which case the next *n* or fewer bytes do not contribute to a complete and valid character. In this case, [EILSEQ] shall be stored in *errno* and the conversion state is undefined.

ERRORS

The *mbrlen*( ) function may fail if:

[EINVAL] — *ps* points to an object that contains an invalid conversion state.

[EILSEQ] — Invalid character sequence is detected.

24253 **EXAMPLES**

24254 None.

24255 **APPLICATION USAGE**

24256 None.

24257 **RATIONALE**

24258 None.

24259 **FUTURE DIRECTIONS**

24260 None.

24261 **SEE ALSO**

24262 *mbsinit*( ), *mbrtowc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

24263 **CHANGE HISTORY**

24264 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
24265 (E).

24266 **Issue 6**

24267 The *mbrlen*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

24268 **NAME**

24269 mbrtowc — convert a character to a wide-character code (restartable)

24270 **SYNOPSIS**

24271 ```
#include <wchar.h>
```

24272 ```
size_t mbrtowc(wchar_t *restrict pwc, const char *restrict s,
```
24273 ```
    size_t n, mbstate_t *restrict ps);
```

24274 **DESCRIPTION**

24275 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24276 conflict between the requirements described here and the ISO C standard is unintentional. This
24277 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24278 If *s* is a null pointer, the *mbrtowc*( ) function shall be equivalent to the call:

24279 ```
mbrtowc(NULL, "", 1, ps)
```

24280 In this case, the values of the arguments *pwc* and *n* are ignored.

24281 If *s* is not a null pointer, the *mbrtowc*( ) function shall inspect at most *n* bytes beginning at the |
24282 byte pointed to by *s* to determine the number of bytes needed to complete the next character
24283 (including any shift sequences). If the function determines that the next character is completed, it |
24284 shall determine the value of the corresponding wide character and then, if *pwc* is not a null |
24285 pointer, shall store that value in the object pointed to by *pwc*. If the corresponding wide |
24286 character is the null wide character, the resulting state described shall be the initial conversion |
24287 state. |

24288 If *ps* is a null pointer, the *mbrtowc*( ) function shall use its own internal **mbstate_t** object, which |
24289 shall be initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** |
24290 object pointed to by *ps* shall be used to completely describe the current conversion state of the |
24291 associated character sequence. The implementation shall behave as if no function defined in this |
24292 volume of IEEE Std 1003.1-200x calls *mbrtowc*( ). |

24293 The behavior of this function is affected by the *LC_CTYPE* category of the current locale. |

24294 **RETURN VALUE**

24295 The *mbrtowc*( ) function shall return the first of the following that applies:

24296 0 If the next *n* or fewer bytes complete the character that corresponds to the null
24297 wide character (which is the value stored). |

24298 between 1 and *n* inclusive |
24299 If the next *n* or fewer bytes complete a valid character (which is the value |
24300 stored); the value returned shall be the number of bytes that complete the
24301 character.

24302 (**size_t**)–2 If the next *n* bytes contribute to an incomplete but potentially valid character,
24303 and all *n* bytes have been processed (no value is stored). When *n* has at least
24304 the value of the {MB_CUR_MAX} macro, this case can only occur if *s* points at
24305 a sequence of redundant shift sequences (for implementations with state-
24306 dependent encodings).

24307 (**size_t**)–1 If an encoding error occurs, in which case the next *n* or fewer bytes do not
24308 contribute to a complete and valid character (no value is stored). In this case,
24309 [EILSEQ] shall be stored in *errno* and the conversion state is undefined.

ERRORS

The *mbrtowc*( ) function may fail if:

CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

[EILSEQ] Invalid character sequence is detected.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbsinit( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The *mbrtowc*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard. |

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The [EINVAL] error condition is added. |

ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated. |

NAME

mbsinit — determine conversion object status

SYNOPSIS

```
#include <wchar.h>

int mbsinit(const mbstate_t *ps);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

If *ps* is not a null pointer, the *mbsinit*( ) function shall determine whether the object pointed to by *ps* describes an initial conversion state.

RETURN VALUE

The *mbsinit*( ) function shall return non-zero if *ps* is a null pointer, or if the pointed-to object describes an initial conversion state; otherwise, it shall return zero.

If an **mbstate_t** object is altered by any of the functions described as ''restartable'', and is then used with a different character sequence, or in the other conversion direction, or with a different *LC_CTYPE* category setting than on earlier function calls, the behavior is undefined.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The **mbstate_t** object is used to describe the current conversion state from a particular character sequence to a wide-character sequence (or *vice versa*) under the rules of a particular setting of the *LC_CTYPE* category of the current locale.

The initial conversion state corresponds, for a conversion in either direction, to the beginning of a new character sequence in the initial shift state. A zero valued **mbstate_t** object is at least one way to describe an initial conversion state. A zero valued **mbstate_t** object can be used to initiate conversion involving any character sequence, in any *LC_CTYPE* category setting.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbrlen( ), *mbrtowc*( ), *wcrtomb*( ), *mbsrtowcs*( ), *wcsrtombs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

NAME

mbsrtowcs — convert a character string to a wide-character string (restartable)

SYNOPSIS

```
#include <wchar.h>

size_t mbsrtowcs(wchar_t *restrict dst, const char **restrict src,
    size_t len, mbstate_t *restrict ps);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *mbsrtowcs*( ) function shall convert a sequence of characters, beginning in the conversion state described by the object pointed to by *ps*, from the array indirectly pointed to by *src* into a sequence of corresponding wide characters. If *dst* is not a null pointer, the converted characters shall be stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null character, which shall also be stored. Conversion shall stop early in either of the following cases:

- A sequence of bytes is encountered that does not form a valid character.
- *len* codes have been stored into the array pointed to by *dst* (and *dst* is not a null pointer).

Each conversion shall take place as if by a call to the *mbrtowc*( ) function.

If *dst* is not a null pointer, the pointer object pointed to by *src* shall be assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last character converted (if any). If conversion stopped due to reaching a terminating null character, and if *dst* is not a null pointer, the resulting state described shall be the initial conversion state.

If *ps* is a null pointer, the *mbsrtowcs*( ) function shall use its own internal **mbstate_t** object, which is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* shall be used to completely describe the current conversion state of the associated character sequence. The implementation behaves as if no function defined in this volume of IEEE Std 1003.1-200x calls *mbsrtowcs*( ).

The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

If the input conversion encounters a sequence of bytes that do not form a valid character, an encoding error occurs. In this case, the *mbsrtowcs*( ) function stores the value of the macro [EILSEQ] in *errno* and shall return (**size_t**)–1; the conversion state is undefined. Otherwise, it shall return the number of characters successfully converted, not including the terminating null (if any).

ERRORS

The *mbsrtowcs*( ) function may fail if:

CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

[EILSEQ] Invalid character sequence is detected.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbsinit( ), *mbrtowc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The *mbsrtowcs*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard. |

The [EINVAL] error condition is marked CX. |

24428 **NAME**

24429 mbstowcs — convert a character string to a wide-character string

24430 **SYNOPSIS**

24431 ```
#include <stdlib.h>
```

24432 ```
size_t mbstowcs(wchar_t *restrict pwcs, const char *restrict s,
```

24433 ```
    size_t n);
```

24434 **DESCRIPTION**

24435 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24436 conflict between the requirements described here and the ISO C standard is unintentional. This
24437 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24438 The *mbstowcs*( ) function shall convert a sequence of characters that begins in the initial shift
24439 state from the array pointed to by *s* into a sequence of corresponding wide-character codes and |
24440 shall store not more than *n* wide-character codes into the array pointed to by *pwcs*. No |
24441 characters that follow a null byte (which is converted into a wide-character code with value 0)
24442 shall be examined or converted. Each character shall be converted as if by a call to *mbtowc*( ), |
24443 except that the shift state of *mbtowc*( ) is not affected.

24444 No more than *n* elements shall be modified in the array pointed to by *pwcs*. If copying takes
24445 place between objects that overlap, the behavior is undefined.

24446 XSI The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale. If
24447 *pwcs* is a null pointer, *mbstowcs*( ) shall return the length required to convert the entire array
24448 regardless of the value of *n*, but no values are stored.

24449 **RETURN VALUE**

24450 CX If an invalid character is encountered, *mbstowcs*( ) shall return (**size_t**)–1 and may set *errno* to
24451 XSI indicate the error. Otherwise, *mbstowcs*( ) shall return the number of the array elements modified |
24452 (or required if *pwcs* is null), not including a terminating 0 code, if any. The array shall not be |
24453 zero-terminated if the value returned is *n*.

24454 **ERRORS**

24455 The *mbstowcs*( ) function may fail if:

24456 XSI [EILSEQ] Invalid byte sequence is detected.

24457 **EXAMPLES**

24458 None.

24459 **APPLICATION USAGE**

24460 None.

24461 **RATIONALE**

24462 None.

24463 **FUTURE DIRECTIONS**

24464 None.

24465 **SEE ALSO**

24466 *mblen*( ), *mbtowc*( ), *wctomb*( ), *wcstombs*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
24467 **<stdlib.h>**

24468 **CHANGE HISTORY**

24469 First released in Issue 4. Aligned with the ISO C standard.
```

24470 **Issue 6**

24471 The *mbstowcs*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard. |

24472 Extensions beyond the ISO C standard are now marked. |

24473 **NAME**

24474 mbtowc — convert a character to a wide-character code

24475 **SYNOPSIS**

24476 `#include <stdlib.h>`

24477 `int mbtowc(wchar_t *restrict pwc, const char *restrict s, size_t n);`

24478 **DESCRIPTION**

24479 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24480 conflict between the requirements described here and the ISO C standard is unintentional. This
24481 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24482 If *s* is not a null pointer, *mbtowc*( ) shall determine the number of the bytes that constitute the
24483 character pointed to by *s*. It shall then determine the wide-character code for the value of type |
24484 **wchar_t** that corresponds to that character. (The value of the wide-character code corresponding
24485 to the null byte is 0.) If the character is valid and *pwc* is not a null pointer, *mbtowc*( ) shall store |
24486 the wide-character code in the object pointed to by *pwc*. |

24487 The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a
24488 state-dependent encoding, this function is placed into its initial state by a call for which its
24489 character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null |
24490 pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a |
24491 null pointer shall cause this function to return a non-zero value if encodings have state |
24492 dependency, and 0 otherwise. If the implementation employs special bytes to change the shift |
24493 state, these bytes shall not produce separate wide-character codes, but shall be grouped with an |
24494 adjacent character. Changing the *LC_CTYPE* category causes the shift state of this function to be |
24495 unspecified. At most *n* bytes of the array pointed to by *s* shall be examined. |

24496 The implementation shall behave as if no function defined in this volume of
24497 IEEE Std 1003.1-200x calls *mbtowc*( ).

24498 **RETURN VALUE**

24499 If *s* is a null pointer, *mbtowc*( ) shall return a non-zero or 0 value, if character encodings,
24500 respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *mbtowc*( )
24501 shall either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the
24502 CX converted character (if the next *n* or fewer bytes form a valid character), or return −1 and may
24503 set *errno* to indicate the error(if they do not form a valid character).

24504 In no case shall the value returned be greater than *n* or the value of the {MB_CUR_MAX} macro.

24505 **ERRORS**

24506 The *mbtowc*( ) function may fail if:

24507 XSI [EILSEQ] Invalid character sequence is detected.

24508 **EXAMPLES**

24509 None.

24510 **APPLICATION USAGE**

24511 None.

24512 **RATIONALE**

24513 None.

24514 **FUTURE DIRECTIONS**

24515 None.

SEE ALSO

mblen( ), *mbstowcs*( ), *wctomb*( ), *wcstombs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Aligned with the ISO C standard.

Issue 6

The *mbtowc*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard. |

Extensions beyond the ISO C standard are now marked. |

24524 **NAME**

24525 memccpy — copy bytes in memory

24526 **SYNOPSIS**

24527 XSI
```
#include <string.h>

void *memccpy(void *restrict s1, const void *restrict s2,
    int c, size_t n);
```
24528
24529
24530

24531 **DESCRIPTION**

24532 The *memccpy*( ) function shall copy bytes from memory area *s2* into *s1*, stopping after the first
24533 occurrence of byte *c* (converted to an **unsigned char**) is copied, or after *n* bytes are copied,
24534 whichever comes first. If copying takes place between objects that overlap, the behavior is
24535 undefined.

24536 **RETURN VALUE**

24537 The *memccpy*( ) function shall return a pointer to the byte after the copy of *c* in *s1*, or a null
24538 pointer if *c* was not found in the first *n* bytes of *s2*.

24539 **ERRORS**

24540 No errors are defined.

24541 **EXAMPLES**

24542 None.

24543 **APPLICATION USAGE**

24544 The *memccpy*( ) function does not check for the overflow of the receiving memory area.

24545 **RATIONALE**

24546 None.

24547 **FUTURE DIRECTIONS**

24548 None.

24549 **SEE ALSO**

24550 The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

24551 **CHANGE HISTORY**

24552 First released in Issue 1. Derived from Issue 1 of the SVID.

24553 **Issue 6**

24554 The **restrict** keyword is added to the *memccpy*( ) prototype for alignment with the
24555 ISO / IEC 9899: 1999 standard.

24556 **NAME**

24557 memchr — find byte in memory

24558 **SYNOPSIS**

24559
```
#include <string.h>
```

24560
```
void *memchr(const void *s, int c, size_t n);
```

24561 **DESCRIPTION**

24562 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24563 conflict between the requirements described here and the ISO C standard is unintentional. This
24564 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24565 The *memchr*( ) function shall locate the first occurrence of *c* (converted to an **unsigned char**) in
24566 the initial *n* bytes (each interpreted as **unsigned char**) of the object pointed to by *s*.

24567 **RETURN VALUE**

24568 The *memchr*( ) function shall return a pointer to the located byte, or a null pointer if the byte does
24569 not occur in the object.

24570 **ERRORS**

24571 No errors are defined.

24572 **EXAMPLES**

24573 None.

24574 **APPLICATION USAGE**

24575 None.

24576 **RATIONALE**

24577 None.

24578 **FUTURE DIRECTIONS**

24579 None.

24580 **SEE ALSO**

24581 The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

24582 **CHANGE HISTORY**

24583 First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

memcmp — compare bytes in memory

SYNOPSIS

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *memcmp*( ) function shall compare the first *n* bytes (each interpreted as **unsigned char**) of the object pointed to by *s1* to the first *n* bytes of the object pointed to by *s2*.

The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the objects being compared.

RETURN VALUE

The *memcmp*( ) function shall return an integer greater than, equal to, or less than 0, if the object pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*, respectively.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

24615 **NAME**

24616 memcpy — copy bytes in memory

24617 **SYNOPSIS**

24618 `#include <string.h>`

24619 `void *memcpy(void *restrict s1, const void *restrict s2, size_t n);`

24620 **DESCRIPTION**

24621 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24622 conflict between the requirements described here and the ISO C standard is unintentional. This
24623 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24624 The *memcpy*( ) function shall copy *n* bytes from the object pointed to by *s2* into the object pointed
24625 to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

24626 **RETURN VALUE**

24627 The *memcpy*( ) function shall return *s1*; no return value is reserved to indicate an error.

24628 **ERRORS**

24629 No errors are defined.

24630 **EXAMPLES**

24631 None.

24632 **APPLICATION USAGE**

24633 The *memcpy*( ) function does not check for the overflowing of the receiving memory area.

24634 **RATIONALE**

24635 None.

24636 **FUTURE DIRECTIONS**

24637 None.

24638 **SEE ALSO**

24639 The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

24640 **CHANGE HISTORY**

24641 First released in Issue 1. Derived from Issue 1 of the SVID.

24642 **Issue 6**

24643 The *memcpy*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

24644 **NAME**

24645 memmove — copy bytes in memory with overlapping areas

24646 **SYNOPSIS**

24647 `#include <string.h>`

24648 `void *memmove(void *s1, const void *s2, size_t n);`

24649 **DESCRIPTION**

24650 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24651 conflict between the requirements described here and the ISO C standard is unintentional. This
24652 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24653 The *memmove*( ) function shall copy *n* bytes from the object pointed to by *s2* into the object
24654 pointed to by *s1*. Copying takes place as if the *n* bytes from the object pointed to by *s2* are first
24655 copied into a temporary array of *n* bytes that does not overlap the objects pointed to by *s1* and
24656 *s2*, and then the *n* bytes from the temporary array are copied into the object pointed to by *s1*.

24657 **RETURN VALUE**

24658 The *memmove*( ) function shall return *s1*; no return value is reserved to indicate an error.

24659 **ERRORS**

24660 No errors are defined.

24661 **EXAMPLES**

24662 None.

24663 **APPLICATION USAGE**

24664 None.

24665 **RATIONALE**

24666 None.

24667 **FUTURE DIRECTIONS**

24668 None.

24669 **SEE ALSO**

24670 The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

24671 **CHANGE HISTORY**

24672 First released in Issue 4. Derived from the ANSI C standard.

24673 **NAME**

24674 memset — set bytes in memory

24675 **SYNOPSIS**

24676
```
#include <string.h>
```

24677
```
void *memset(void *s, int c, size_t n);
```

24678 **DESCRIPTION**

24679 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24680 conflict between the requirements described here and the ISO C standard is unintentional. This
24681 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24682 The *memset*( ) function shall copy *c* (converted to an **unsigned char**) into each of the first *n* bytes
24683 of the object pointed to by *s*.

24684 **RETURN VALUE**

24685 The *memset*( ) function shall return *s*; no return value is reserved to indicate an error.

24686 **ERRORS**

24687 No errors are defined.

24688 **EXAMPLES**

24689 None.

24690 **APPLICATION USAGE**

24691 None.

24692 **RATIONALE**

24693 None.

24694 **FUTURE DIRECTIONS**

24695 None.

24696 **SEE ALSO**

24697 The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

24698 **CHANGE HISTORY**

24699 First released in Issue 1. Derived from Issue 1 of the SVID.

24700 **NAME**

24701 mkdir — make a directory

24702 **SYNOPSIS**

24703 `#include <sys/stat.h>`

24704 `int mkdir(const char *path, mode_t mode);`

24705 **DESCRIPTION**

24706 The *mkdir*( ) function shall create a new directory with name *path*. The file permission bits of the |
24707 new directory shall be initialized from *mode*. These file permission bits of the *mode* argument |
24708 shall be modified by the process' file creation mask. |

24709 When bits in *mode* other than the file permission bits are set, the meaning of these additional bits
24710 is implementation-defined.

24711 The directory's user ID shall be set to the process' effective user ID. The directory's group ID |
24712 shall be set to the group ID of the parent directory or to the effective group ID of the process. |
24713 Implementations shall provide a way to initialize the directory's group ID to the group ID of the |
24714 parent directory. Implementations may, but need not, provide an implementation-defined way |
24715 to initialize the directory's group ID to the effective group ID of the calling process. |

24716 The newly created directory shall be an empty directory.

24717 If *path* names a symbolic link, *mkdir*( ) shall fail and set *errno* to [EEXIST].

24718 Upon successful completion, *mkdir*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
24719 fields of the directory. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the
24720 new entry shall be marked for update.

24721 **RETURN VALUE**

24722 Upon successful completion, *mkdir*( ) shall return 0. Otherwise, –1 shall be returned, no directory
24723 shall be created, and *errno* shall be set to indicate the error.

24724 **ERRORS**

24725 The *mkdir*( ) function shall fail if:

24726 [EACCES] Search permission is denied on a component of the path prefix, or write
24727 permission is denied on the parent directory of the directory to be created.

24728 [EEXIST] The named file exists.

24729 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
24730 argument.

24731 [EMLINK] The link count of the parent directory would exceed {LINK_MAX}.

24732 [ENAMETOOLONG]
24733 The length of the *path* argument exceeds {PATH_MAX} or a pathname |
24734 component is longer than {NAME_MAX}. |

24735 [ENOENT] A component of the path prefix specified by *path* does not name an existing
24736 directory or *path* is an empty string.

24737 [ENOSPC] The file system does not contain enough space to hold the contents of the new
24738 directory or to extend the parent directory of the new directory.

24739 [ENOTDIR] A component of the path prefix is not a directory.

24740 [EROFS] The parent directory resides on a read-only file system.

The *mkdir*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

EXAMPLES

Creating a Directory

The following example shows how to create a directory named **/home/cnd/mod1**, with read/write/search permissions for owner and group, and with read/search permissions for others.

```
#include <sys/types.h>
#include <sys/stat.h>

int status;
...
status = mkdir("/home/cnd/mod1", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
```

APPLICATION USAGE

None.

RATIONALE

The *mkdir*( ) function originated in 4.2 BSD and was added to System V in Release 3.0.

4.3 BSD detects [ENAMETOOLONG].

The POSIX.1-1990 standard required that the group ID of a newly created directory be set to the group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required that implementations provide a way to have the group ID be set to the group ID of the containing directory, but did not prohibit implementations also supporting a way to set the group ID to the effective group ID of the creating process. Conforming applications should not assume which group ID will be used. If it matters, an application can use *chown*( ) to set the group ID after the directory is created, or determine under what conditions the implementation will set the desired group ID.

FUTURE DIRECTIONS

None.

SEE ALSO

umask( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

24788 **NAME**

24789 mkfifo — make a FIFO special file

24790 **SYNOPSIS**

24791 `#include <sys/stat.h>`

24792 `int mkfifo(const char *path, mode_t mode);`

24793 **DESCRIPTION**

24794 The *mkfifo*( ) function shall create a new FIFO special file named by the pathname pointed to by |
24795 *path*. The file permission bits of the new FIFO shall be initialized from *mode*. The file permission |
24796 bits of the *mode* argument shall be modified by the process' file creation mask. |

24797 When bits in *mode* other than the file permission bits are set, the effect is implementation-
24798 defined.

24799 If *path* names a symbolic link, *mkfifo*( ) shall fail and set *errno* to [EEXIST].

24800 The FIFO's user ID shall be set to the process' effective user ID. The FIFO's group ID shall be set
24801 to the group ID of the parent directory or to the effective group ID of the process. |
24802 Implementation shall provide a way to initialize the FIFO's group ID to the group ID of the |
24803 parent directory. Implementations may, but need not, provide an implementation-defined way |
24804 to initialize the FIFO's group ID to the effective group ID of the calling process. |

24805 Upon successful completion, *mkfifo*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
24806 fields of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new
24807 entry shall be marked for update.

24808 **RETURN VALUE**

24809 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned, no FIFO shall
24810 be created, and *errno* shall be set to indicate the error.

24811 **ERRORS**

24812 The *mkfifo*( ) function shall fail if:

24813 [EACCES] A component of the path prefix denies search permission, or write permission
24814 is denied on the parent directory of the FIFO to be created.

24815 [EEXIST] The named file already exists.

24816 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
24817 argument.

24818 [ENAMETOOLONG]
24819 The length of the *path* argument exceeds {PATH_MAX} or a pathname |
24820 component is longer than {NAME_MAX}. |

24821 [ENOENT] A component of the path prefix specified by *path* does not name an existing
24822 directory or *path* is an empty string.

24823 [ENOSPC] The directory that would contain the new file cannot be extended or the file
24824 system is out of file-allocation resources.

24825 [ENOTDIR] A component of the path prefix is not a directory.

24826 [EROFS] The named file resides on a read-only file system.

24827 The *mkfifo*( ) function may fail if:

24828 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
24829 resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

EXAMPLES

Creating a FIFO File

The following example shows how to create a FIFO file named **/home/cnd/mod_done**, with read/write permissions for owner, and with read permissions for group and others.

```
#include <sys/types.h>
#include <sys/stat.h>

int status;
...
status = mkfifo("/home/cnd/mod_done", S_IWUSR | S_IRUSR |
    S_IRGRP | S_IROTH);
```

APPLICATION USAGE

None.

RATIONALE

The syntax of this function is intended to maintain compatibility with historical implementations of *mknod*(). The latter function was included in the 1984 /usr/group standard but only for use in creating FIFO special files. The *mknod*() function was originally excluded from the POSIX.1-1988 standard as implementation-defined and replaced by *mkdir*() and *mkfifo*(). The *mknod*() function is now included for alignment with the Single UNIX Specification.

The POSIX.1-1990 standard required that the group ID of a newly created FIFO be set to the group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required that implementations provide a way to have the group ID be set to the group ID of the containing directory, but did not prohibit implementations also supporting a way to set the group ID to the effective group ID of the creating process. Conforming applications should not assume which group ID will be used. If it matters, an application can use *chown*() to set the group ID after the FIFO is created, or determine under what conditions the implementation will set the desired group ID.

FUTURE DIRECTIONS

None.

SEE ALSO

umask(), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

24878 **NAME**

24879 mknod — make a directory, a special or regular file

24880 **SYNOPSIS**

24881 XSI
```
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, dev_t dev);
```
24882

24883

24884 **DESCRIPTION**

24885 The *mknod*( ) function shall create a new file named by the pathname to which the argument *path* |
24886 points.

24887 The file type for *path* is OR'ed into the *mode* argument, and the application shall select one of the
24888 following symbolic constants:

24889

24890

Name	Description
S_IFIFO	FIFO-special
S_IFCHR	Character-special (non-portable)
S_IFDIR	Directory (non-portable)
S_IFBLK	Block-special (non-portable)
S_IFREG	Regular (non-portable)

24895

24896 The only portable use of *mknod*( ) is to create a FIFO-special file. If *mode* is not S_IFIFO or *dev* is
24897 not 0, the behavior of *mknod*( ) is unspecified.

24898 The permissions for the new file are OR'ed into the *mode* argument, and may be selected from
24899 any combination of the following symbolic constants:

24900

24901

Name	Description
S_ISUID	Set user ID on execution.
S_ISGID	Set group ID on execution.
S_IRWXU	Read, write, or execute (search) by owner.
S_IRUSR	Read by owner.
S_IWUSR	Write by owner.
S_IXUSR	Execute (search) by owner.
S_IRWXG	Read, write, or execute (search) by group.
S_IRGRP	Read by group.
S_IWGRP	Write by group.
S_IXGRP	Execute (search) by group.
S_IRWXO	Read, write, or execute (search) by others.
S_IROTH	Read by others.
S_IWOTH	Write by others.
S_IXOTH	Execute (search) by others.
S_ISVTX	On directories, restricted deletion flag.

24916

24917 The user ID of the file shall be initialized to the effective user ID of the process. The group ID of |
24918 the file shall be initialized to either the effective group ID of the process or the group ID of the |
24919 parent directory. Implementations shall provide a way to initialize the file's group ID to the |
24920 group ID of the parent directory. Implementations may, but need not, provide an |
24921 implementation-defined way to initialize the file's gorup ID to the effective group ID of the |
24922 calling proces. |

The owner, group, and other permission bits of *mode* shall be modified by the file mode creation mask of the process. The *mknod*( ) function shall clear each bit whose corresponding bit in the file mode creation mask of the process is set.

If *path* names a symbolic link, *mknod*( ) shall fail and set *errno* to [EEXIST].

Upon successful completion, *mknod*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry shall be marked for update.

Only a process with appropriate privileges may invoke *mknod*( ) for file types other than FIFO-special.

RETURN VALUE

Upon successful completion, *mknod*( ) shall return 0. Otherwise, it shall return –1, the new file shall not be created, and *errno* shall be set to indicate the error.

ERRORS

The *mknod*( ) function shall fail if:

[EACCES] A component of the path prefix denies search permission, or write permission is denied on the parent directory.

[EEXIST] The named file exists.

[EINVAL] An invalid argument exists.

[EIO] An I/O error occurred while accessing the file system.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[ENAMETOOLONG] The length of a pathname exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

[ENOENT] A component of the path prefix specified by *path* does not name an existing directory or *path* is an empty string.

[ENOSPC] The directory that would contain the new file cannot be extended or the file system is out of file allocation resources.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The invoking process does not have appropriate privileges and the file type is not FIFO-special.

[EROFS] The directory in which the file is to be created is located on a read-only file system.

The *mknod*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

EXAMPLES

Creating a FIFO Special File

The following example shows how to create a FIFO special file named **/home/cnd/mod_done**, with read/write permissions for owner, and with read permissions for group and others.

```
#include <sys/types.h>
#include <sys/stat.h>

dev_t dev;
int   status;
...
status  = mknod("/home/cnd/mod_done", S_IFIFO | S_IWUSR |
    S_IRUSR | S_IRGRP | S_IROTH, dev);
```

APPLICATION USAGE

mkfifo( ) is preferred over this function for making FIFO special files.

RATIONALE

The POSIX.1-1990 standard required that the group ID of a newly created file be set to the group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required that implementations provide a way to have the group ID be set to the group ID of the containing directory, but did not prohibit implementations also supporting a way to set the group ID to the effective group ID of the creating process. Conforming applications should not assume which group ID will be used. If it matters, an application can use *chown*( ) to set the group ID after the file is created, or determine under what conditions the implementation will set the desired group ID.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *creat*( ), *exec*, *mkdir*( ), *mkfifo*( ), *open*( ), *stat*( ), *umask*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

mkstemp — make a unique filename

SYNOPSIS

XSI

```
#include <stdlib.h>

int mkstemp(char *template);
```

DESCRIPTION

The *mkstemp*( ) function shall replace the contents of the string pointed to by *template* by a unique filename, and return a file descriptor for the file open for reading and writing. The function thus prevents any possible race condition between testing whether the file exists and opening it for use. The string in *template* should look like a filename with six trailing *X*s; *mkstemp*( ) replaces each *X* with a character from the portable filename character set. The characters are chosen such that the resulting name does not duplicate the name of an existing file at the time of a call to *mkstemp*( ).

RETURN VALUE

Upon successful completion, *mkstemp*( ) shall return an open file descriptor. Otherwise, –1 shall be returned if no suitable file could be created.

ERRORS

No errors are defined.

EXAMPLES

Generating a Filename

The following example creates a file with a 10-character name beginning with the characters `"file"` and opens the file for reading and writing. The value returned as the value of *fd* is a file descriptor that identifies the file.

```
#include <stdlib.h>
...
char template[] = "/tmp/fileXXXXXX";
int fd;

fd = mkstemp(template);
```

APPLICATION USAGE

It is possible to run out of letters.

The *mkstemp*( ) function need not check to determine whether the filename part of *template* exceeds the maximum allowable filename length.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getpid( ), *open*( ), *tmpfile*( ), *tmpnam*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

25037 **CHANGE HISTORY**

25038 First released in Issue 4, Version 2.

25039 **Issue 5**

25040 Moved from X/OPEN UNIX extension to BASE.

25041 **NAME**

25042 mktemp — make a unique filename (**LEGACY**)

25043 **SYNOPSIS**

25044 XSI
```
#include <stdlib.h>

char *mktemp(char *template);
```
25045

25046

25047 **DESCRIPTION**

25048 The *mktemp*( ) function shall replace the contents of the string pointed to by *template* by a unique
25049 filename and return *template*. The application shall initialize *template* to be a filename with six
25050 trailing *X*s; *mktemp*( ) shall replace each *X* with a single byte character from the portable filename
25051 character set.

25052 **RETURN VALUE**

25053 The *mktemp*( ) function shall return the pointer *template*. If a unique name cannot be created,
25054 *template* shall point to a null string.

25055 **ERRORS**

25056 No errors are defined.

25057 **EXAMPLES**

25058 **Generating a Filename**

25059 The following example replaces the contents of the `"template"` string with a 10-character
25060 filename beginning with the characters `"file"` and returns a pointer to the `"template"` string
25061 that contains the new filename.

```
25062 #include <stdlib.h>
25063 ...
25064 char *template = "/tmp/fileXXXXXX";
25065 char *ptr;

25066 ptr = mktemp(template);
```

25067 **APPLICATION USAGE**

25068 Between the time a pathname is created and the file opened, it is possible for some other process |
25069 to create a file with the same name. The *mkstemp*( ) function avoids this problem and is preferred
25070 over this function.

25071 **RATIONALE**

25072 None.

25073 **FUTURE DIRECTIONS**

25074 This function may be withdrawn in a future version.

25075 **SEE ALSO**

25076 *mkstemp*( ), *tmpfile*( ), *tmpnam*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

25077 **CHANGE HISTORY**

25078 First released in Issue 4, Version 2.

25079 **Issue 5**

25080 Moved from X/OPEN UNIX extension to BASE.

25081 **Issue 6**

25082 This function is marked LEGACY.

25083 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

25084 **NAME**

25085 mktime — convert broken-down time into time since the Epoch

25086 **SYNOPSIS**

25087
```
#include <time.h>
```

25088
```
time_t mktime(struct tm *timeptr);
```

25089 **DESCRIPTION**

25090 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25091 conflict between the requirements described here and the ISO C standard is unintentional. This
25092 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25093 The *mktime*( ) function shall convert the broken-down time, expressed as local time, in the
25094 structure pointed to by *timeptr*, into a time since the Epoch value with the same encoding as that
25095 of the values returned by *time*( ). The original values of the *tm_wday* and *tm_yday* components of
25096 the structure are ignored, and the original values of the other components are not restricted to
25097 the ranges described in **<time.h>**.

25098 CX A positive or 0 value for *tm_isdst* shall cause *mktime*( ) to presume initially that Daylight Savings
25099 Time, respectively, is or is not in effect for the specified time. A negative value for *tm_isdst* shall
25100 cause *mktime*( ) to attempt to determine whether Daylight Saving Time is in effect for the
25101 specified time.

25102 Local timezone information shall be set as though *mktime*( ) called *tzset*( ). |

25103 The relationship between the **tm** structure (defined in the **<time.h>** header) and the time in |
25104 seconds since the Epoch is that the result shall be as specified in the expression given in the |
25105 definition of seconds since the Epoch (see the Base Definitions volume of IEEE Std 1003.1-200x, |
25106 Section 4.14, Seconds Since the Epoch) corrected for timezone and any seasonal time |
25107 adjustments, where the names in the structure and in the expression correspond. |

25108 Upon successful completion, the values of the *tm_wday* and *tm_yday* components of the structure
25109 shall be set appropriately, and the other components are set to represent the specified time since
25110 the Epoch, but with their values forced to the ranges indicated in the **<time.h>** entry; the final
25111 value of *tm_mday* shall not be set until *tm_mon* and *tm_year* are determined.

25112 **RETURN VALUE**

25113 The *mktime*( ) function shall return the specified time since the Epoch encoded as a value of type
25114 **time_t**. If the time since the Epoch cannot be represented, the function shall return the value
25115 (**time_t**)–1.

25116 **ERRORS**

25117 No errors are defined.

25118 **EXAMPLES**

25119 What day of the week is July 4, 2001?

25120
```
#include <stdio.h>
#include <time.h>
```

25122
```
struct tm time_str;
```

25123
```
char daybuf[20];
```

25124
```
int main(void)
{
    time_str.tm_year = 2001 – 1900;
    time_str.tm_mon = 7 – 1;
    time_str.tm_mday = 4;
```

```
25129           time_str.tm_hour = 0;
25130           time_str.tm_min = 0;
25131           time_str.tm_sec = 1;
25132           time_str.tm_isdst = –1;
25133           if (mktime(&time_str) == -1)
25134               (void)puts("-unknown-");
25135           else {
25136               (void)strftime(daybuf, sizeof(daybuf), "%A", &time_str);
25137               (void)puts(daybuf);
25138           }
25139           return 0;
25140       }
```

25141 **APPLICATION USAGE**

25142 None.

25143 **RATIONALE**

25144 None.

25145 **FUTURE DIRECTIONS**

25146 None.

25147 **SEE ALSO**

25148 *asctime*( ), *clock*( ), *ctime*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ),
25149 the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

25150 **CHANGE HISTORY**

25151 First released in Issue 3.

25152 Entry included for alignment with the POSIX.1-1988 standard and the ANSI C standard.

25153 **Issue 6**

25154 Extensions beyond the ISO C standard are now marked.

25155 **NAME**

25156 mlock, munlock — lock or unlock a range of process address space (**REALTIME**)

25157 **SYNOPSIS**

25158 MLR
```
#include <sys/mman.h>

int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```
25159
25160
25161

25162 **DESCRIPTION**

25163 The *mlock*( ) function shall cause those whole pages containing any part of the address space of
25164 the process starting at address *addr* and continuing for *len* bytes to be memory-resident until
25165 unlocked or until the process exits or *exec*s another process image. The implementation may
25166 require that *addr* be a multiple of {PAGESIZE}.

25167 The *munlock*( ) function shall unlock those whole pages containing any part of the address space
25168 of the process starting at address *addr* and continuing for *len* bytes, regardless of how many
25169 times *mlock*( ) has been called by the process for any of the pages in the specified range. The
25170 implementation may require that *addr* be a multiple of {PAGESIZE}. |

25171 If any of the pages in the range specified to a call to *munlock*( ) are also mapped into the address
25172 spaces of other processes, any locks established on those pages by another process are
25173 unaffected by the call of this process to *munlock*( ). If any of the pages in the range specified by a
25174 call to *munlock*( ) are also mapped into other portions of the address space of the calling process
25175 outside the range specified, any locks established on those pages via the other mappings are also
25176 unaffected by this call.

25177 Upon successful return from *mlock*( ), pages in the specified range shall be locked and memory-
25178 resident. Upon successful return from *munlock*( ), pages in the specified range shall be unlocked
25179 with respect to the address space of the process. Memory residency of unlocked pages is
25180 unspecified.

25181 The appropriate privilege is required to lock process memory with *mlock*( ).

25182 **RETURN VALUE**

25183 Upon successful completion, the *mlock*( ) and *munlock*( ) functions shall return a value of zero.
25184 Otherwise, no change is made to any locks in the address space of the process, and the function
25185 shall return a value of –1 and set *errno* to indicate the error.

25186 **ERRORS**

25187 The *mlock*( ) and *munlock*( ) functions shall fail if:

25188 [ENOMEM] Some or all of the address range specified by the *addr* and *len* arguments does
25189 not correspond to valid mapped pages in the address space of the process.

25190 The *mlock*( ) function shall fail if:

25191 [EAGAIN] Some or all of the memory identified by the operation could not be locked
25192 when the call was made.

25193 The *mlock*( ) and *munlock*( ) functions may fail if:

25194 [EINVAL] The *addr* argument is not a multiple of {PAGESIZE}.

25195 The *mlock*( ) function may fail if:

25196 [ENOMEM] Locking the pages mapped by the specified range would exceed an
25197 implementation-defined limit on the amount of memory that the process may
25198 lock.

25199 [EPERM] The calling process does not have the appropriate privilege to perform the
25200 requested operation.

25201 **EXAMPLES**

25202 None.

25203 **APPLICATION USAGE**

25204 None.

25205 **RATIONALE**

25206 None.

25207 **FUTURE DIRECTIONS**

25208 None.

25209 **SEE ALSO**

25210 *exec*, *exit*( ), *fork*( ), *mlockall*( ), *munmap*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
25211 **<sys/mman.h>**

25212 **CHANGE HISTORY**

25213 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25214 **Issue 6**

25215 The *mlock*( ) and *munlock*( ) functions are marked as part of the Range Memory Locking option.

25216 The [ENOSYS] error condition has been removed as stubs need not be provided if an
25217 implementation does not support the Range Memory Locking option.

25218 **NAME**

25219 mlockall, munlockall — lock/unlock the address space of a process (**REALTIME**)

25220 **SYNOPSIS**

25221 ML
```
#include <sys/mman.h>

int mlockall(int flags);
int munlockall(void);
```
25224

25225 **DESCRIPTION**

25226 The *mlockall*( ) function shall cause all of the pages mapped by the address space of a process to
25227 be memory-resident until unlocked or until the process exits or *exec*s another process image. The
25228 *flags* argument determines whether the pages to be locked are those currently mapped by the
25229 address space of the process, those that are mapped in the future, or both. The *flags* argument is
25230 constructed from the bitwise-inclusive OR of one or more of the following symbolic constants,
25231 defined in **<sys/mman.h>**:

25232 MCL_CURRENT Lock all of the pages currently mapped into the address space of the process.

25233 MCL_FUTURE Lock all of the pages that become mapped into the address space of the
25234 process in the future, when those mappings are established.

25235 If MCL_FUTURE is specified, and the automatic locking of future mappings eventually causes
25236 the amount of locked memory to exceed the amount of available physical memory or any other
25237 implementation-defined limit, the behavior is implementation-defined. The manner in which the
25238 implementation informs the application of these situations is also implementation-defined.

25239 The *munlockall*( ) function shall unlock all currently mapped pages of the address space of the |
25240 process. Any pages that become mapped into the address space of the process after a call to |
25241 *munlockall*( ) shall not be locked, unless there is an intervening call to *mlockall*( ) specifying
25242 MCL_FUTURE or a subsequent call to *mlockall*( ) specifying MCL_CURRENT. If pages mapped
25243 into the address space of the process are also mapped into the address spaces of other processes
25244 and are locked by those processes, the locks established by the other processes shall be |
25245 unaffected by a call by this process to *munlockall*( ). |

25246 Upon successful return from the *mlockall*( ) function that specifies MCL_CURRENT, all currently
25247 mapped pages of the process' address space shall be memory-resident and locked. Upon return
25248 from the *munlockall*( ) function, all currently mapped pages of the process' address space shall be
25249 unlocked with respect to the process' address space. The memory residency of unlocked pages is
25250 unspecified.

25251 The appropriate privilege is required to lock process memory with *mlockall*( ).

25252 **RETURN VALUE**

25253 Upon successful completion, the *mlockall*( ) function shall return a value of zero. Otherwise, no
25254 additional memory shall be locked, and the function shall return a value of –1 and set *errno* to
25255 indicate the error. The effect of failure of *mlockall*( ) on previously existing locks in the address
25256 space is unspecified.

25257 If it is supported by the implementation, the *munlockall*( ) function shall always return a value of
25258 zero. Otherwise, the function shall return a value of –1 and set *errno* to indicate the error.

25259 **ERRORS**

25260 The *mlockall*( ) function shall fail if:

25261 [EAGAIN] Some or all of the memory identified by the operation could not be locked
25262 when the call was made.

[EINVAL] The *flags* argument is zero, or includes unimplemented flags.

The *mlockall*( ) function may fail if:

[ENOMEM] Locking all of the pages currently mapped into the address space of the process would exceed an implementation-defined limit on the amount of memory that the process may lock.

[EPERM] The calling process does not have the appropriate privilege to perform the requested operation.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit*( ), *fork*( ), *mlock*( ), *munmap*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mlockall*( ) and *munlockall*( ) functions are marked as part of the Process Memory Locking option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Memory Locking option.

25288 **NAME**

25289 mmap — map pages of memory

25290 **SYNOPSIS**

25291 MF|SHM
```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags,
    int fildes, off_t off);
```
25294

25295 **DESCRIPTION**

25296 The *mmap*( ) function shall establish a mapping between a process' address space and a file,
25297 TYM shared memory object, or typed memory object. The format of the call is as follows:

25298 `pa=mmap(addr, len, prot, flags, fildes, off);`

25299 The *mmap*( ) function shall establish a mapping between the address space of the process at an |
25300 address *pa* for *len* bytes to the memory object represented by the file descriptor *fildes* at offset *off* |
25301 for *len* bytes. The value of *pa* is an implementation-defined function of the parameter *addr* and
25302 the values of *flags*, further described below. A successful *mmap*( ) call shall return *pa* as its result.
25303 The address range starting at *pa* and continuing for *len* bytes shall be legitimate for the possible
25304 (not necessarily current) address space of the process. The range of bytes starting at *off* and
25305 continuing for *len* bytes shall be legitimate for the possible (not necessarily current) offsets in the
25306 TYM file, shared memory object, or typed memory object represented by *fildes*.

25307 TYM If *fildes* represents a typed memory object opened with either the
25308 POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG
25309 flag, the memory object to be mapped shall be that portion of the typed memory object allocated
25310 by the implementation as specified below. In this case, if *off* is non-zero, the behavior of *mmap*( )
25311 is undefined. If *fildes* refers to a valid typed memory object that is not accessible from the calling
25312 process, *mmap*( ) shall fail.

25313 The mapping established by *mmap*( ) shall replace any previous mappings for those whole pages |
25314 containing any part of the address space of the process starting at *pa* and continuing for *len* |
25315 bytes.

25316 If the size of the mapped file changes after the call to *mmap*( ) as a result of some other operation
25317 on the mapped file, the effect of references to portions of the mapped region that correspond to
25318 added or removed portions of the file is unspecified.

25319 TYM The *mmap*( ) function shall be supported for regular files, shared memory objects, and typed |
25320 memory objects. Support for any other type of file is unspecified.

25321 The parameter *prot* determines whether read, write, execute, or some combination of accesses
25322 are permitted to the data being mapped. The *prot* shall be either PROT_NONE or the bitwise- |
25323 inclusive OR of one or more of the other flags in the following table, defined in the
25324 **<sys/mman.h>** header.

25325
25326

Symbolic Constant	Description
PROT_READ	Data can be read.
PROT_WRITE	Data can be written.
PROT_EXEC	Data can be executed.
PROT_NONE	Data cannot be accessed.

25327
25328
25329
25330

25331 If an implementation cannot support the combination of access types specified by *prot*, the call
25332 MPR to *mmap*( ) shall fail. An implementation may permit accesses other than those specified by *prot*; |
25333 however, if the Memory Protection option is supported, the implementation shall not permit a

write to succeed where PROT_WRITE has not been set or shall not permit any access where PROT_NONE alone has been set. The implementation shall support at least the following values of *prot*: PROT_NONE, PROT_READ, PROT_WRITE, and the bitwise-inclusive OR of PROT_READ and PROT_WRITE. If the Memory Protection option is not supported, the result of any access that conflicts with the specified protection is undefined. The file descriptor *fildes* shall have been opened with read permission, regardless of the protection options specified. If PROT_WRITE is specified, the application shall ensure that it has opened the file descriptor *fildes* with write permission unless MAP_PRIVATE is specified in the *flags* parameter as described below.

The parameter *flags* provides other information about the handling of the mapped data. The value of *flags* is the bitwise-inclusive OR of these options, defined in **<sys/mman.h>**:

Symbolic Constant	Description
MAP_SHARED	Changes are shared.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret *addr* exactly.

Implementations that do not support the Memory Mapped Files option are not required to support MAP_PRIVATE.

XSI It is implementation-defined whether MAP_FIXED shall be supported. MAP_FIXED shall be supported on XSI-conformant systems.

MAP_SHARED and MAP_PRIVATE describe the disposition of write references to the memory object. If MAP_SHARED is specified, write references shall change the underlying object. If MAP_PRIVATE is specified, modifications to the mapped data by the calling process shall be visible only to the calling process and shall not change the underlying object. It is unspecified whether modifications to the underlying object done after the MAP_PRIVATE mapping is established are visible through the MAP_PRIVATE mapping. Either MAP_SHARED or MAP_PRIVATE can be specified, but not both. The mapping type is retained across *fork*( ).

TYM When *fildes* represents a typed memory object opened with either the POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, *mmap*( ) shall, if there are enough resources available, map *len* bytes allocated from the corresponding typed memory object which were not previously allocated to any process in any processor that may access that typed memory object. If there are not enough resources available, the function shall fail. If *fildes* represents a typed memory object opened with the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, these allocated bytes shall be contiguous within the typed memory object. If *fildes* represents a typed memory object opened with the POSIX_TYPED_MEM_ALLOCATE flag, these allocated bytes may be composed of non-contiguous fragments within the typed memory object. If *fildes* represents a typed memory object opened with neither the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag nor the POSIX_TYPED_MEM_ALLOCATE flag, *len* bytes starting at offset *off* within the typed memory object are mapped, exactly as when mapping a file or shared memory object. In this case, if two processes map an area of typed memory using the same *off* and *len* values and using file descriptors that refer to the same memory pool (either from the same port or from a different port), both processes shall map the same region of storage.

When MAP_FIXED is set in the *flags* argument, the implementation is informed that the value of *pa* shall be *addr*, exactly. If MAP_FIXED is set, *mmap*( ) may return MAP_FAILED and set *errno* to [EINVAL]. If a MAP_FIXED request is successful, the mapping established by *mmap*( ) replaces any previous mappings for the process' pages in the range [*pa*,*pa*+*len*).

When MAP_FIXED is not set, the implementation uses *addr* in an implementation-defined manner to arrive at *pa*. The *pa* so chosen shall be an area of the address space that the implementation deems suitable for a mapping of *len* bytes to the file. All implementations interpret an *addr* value of 0 as granting the implementation complete freedom in selecting *pa*, subject to constraints described below. A non-zero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the implementation selects a value for *pa*, it never places a mapping at address 0, nor does it replace any extant mapping.

The *off* argument is constrained to be aligned and sized according to the value returned by *sysconf*( ) when passed _SC_PAGESIZE or _SC_PAGE_SIZE. When MAP_FIXED is specified, the application shall ensure that the argument *addr* also meets these constraints. The implementation performs mapping operations over whole pages. Thus, while the argument *len* need not meet a size or alignment constraint, the implementation shall include, in any mapping operation, any partial page specified by the range [*pa*,*pa*+*len*).

The system shall always zero-fill any partial page at the end of an object. Further, the system shall never write out any modified portions of the last page of an object which are beyond its end. MPR References within the address range starting at *pa* and continuing for *len* bytes to whole pages following the end of an object shall result in delivery of a SIGBUS signal.

An implementation may generate SIGBUS signals when a reference would cause an error in the mapped object, such as out-of-space condition.

The *mmap*( ) function shall add an extra reference to the file associated with the file descriptor *fildes* which is not removed by a subsequent *close*( ) on that file descriptor. This reference shall be removed when there are no more mappings to the file.

The *st_atime* field of the mapped file may be marked for update at any time between the *mmap*( ) call and the corresponding *munmap*( ) call. The initial read or write reference to a mapped region shall cause the file's *st_atime* field to be marked for update if it has not already been marked for update.

The *st_ctime* and *st_mtime* fields of a file that is mapped with MAP_SHARED and PROT_WRITE shall be marked for update at some point in the interval between a write reference to the mapped region and the next call to *msync*( ) with MS_ASYNC or MS_SYNC for that portion of the file by any process. If there is no such call and if the underlying file is modified as a result of a write reference, then these fields shall be marked for update at some time after the write reference.

There may be implementation-defined limits on the number of memory regions that can be mapped (per process or per system).

XSI If such a limit is imposed, whether the number of memory regions that can be mapped by a process is decreased by the use of *shmat*( ) is implementation-defined.

If *mmap*( ) fails for reasons other than [EBADF], [EINVAL], or [ENOTSUP], some of the mappings in the address range starting at *addr* and continuing for *len* bytes may have been unmapped.

RETURN VALUE

Upon successful completion, the *mmap*( ) function shall return the address at which the mapping was placed (*pa*); otherwise, it shall return a value of MAP_FAILED and set *errno* to indicate the error. The symbol MAP_FAILED is defined in the **<sys/mman.h>** header. No successful return from *mmap*( ) shall return the value MAP_FAILED.

25425 **ERRORS**

25426 The *mmap*( ) function shall fail if:

25427 [EACCES] The *fildes* argument is not open for read, regardless of the protection specified,
25428 or *fildes* is not open for write and PROT_WRITE was specified for a
25429 MAP_SHARED type mapping.

25430 ML [EAGAIN] The mapping could not be locked in memory, if required by *mlockall*( ), due to
25431 a lack of resources.

25432 [EBADF] The *fildes* argument is not a valid open file descriptor.

25433 [EINVAL] The *addr* argument (if MAP_FIXED was specified) or *off* is not a multiple of
25434 the page size as returned by *sysconf*( ), or are considered invalid by the
25435 implementation.

25436 [EINVAL] The value of *flags* is invalid (neither MAP_PRIVATE nor MAP_SHARED is
25437 set).

25438 [EMFILE] The number of mapped regions would exceed an implementation-defined
25439 limit (per process or per system).

25440 [ENODEV] The *fildes* argument refers to a file whose type is not supported by *mmap*( ).

25441 [ENOMEM] MAP_FIXED was specified, and the range [*addr*,*addr*+*len*) exceeds that allowed
25442 for the address space of a process; or, if MAP_FIXED was not specified and
25443 there is insufficient room in the address space to effect the mapping.

25444 ML [ENOMEM] The mapping could not be locked in memory, if required by *mlockall*( ),
25445 because it would require more space than the system is able to supply.

25446 MAP_FIXED or MAP_PRIVATE was specified in the *flags* argument and the
25447 implementation does not support this functionality.

25448 TYM [ENOMEM] Not enough unallocated memory resources remain in the typed memory
25449 object designated by *fildes* to allocate *len* bytes.

25450 [ENOTSUP] The implementation does not support the combination of accesses requested
25451 in the *prot* argument.

25452 [ENXIO] Addresses in the range [*off*,*off*+*len*) are invalid for the object specified by *fildes*.

25453 [ENXIO] MAP_FIXED was specified in *flags* and the combination of *addr*, *len*, and *off* is
25454 invalid for the object specified by *fildes*.

25455 TYM [ENXIO] The *fildes* argument refers to a typed memory object that is not accessible from
25456 the calling process.

25457 [EOVERFLOW] The file is a regular file and the value of *off* plus *len* exceeds the offset
25458 maximum established in the open file description associated with *fildes*.

25459 **EXAMPLES**

25460 None.

25461 **APPLICATION USAGE**

25462 Use of *mmap*( ) may reduce the amount of memory available to other memory allocation
25463 functions.

25464 Use of MAP_FIXED may result in unspecified behavior in further use of *malloc*( ) and *shmat*( ).
25465 The use of MAP_FIXED is discouraged, as it may prevent an implementation from making the
25466 most effective use of resources.

The application must ensure correct synchronization when using *mmap*( ) in conjunction with any other file access method, such as *read*( ) and *write*( ), standard input/output, and *shmat*( ).

The *mmap*( ) function allows access to resources via address space manipulations, instead of *read*( )/*write*( ). Once a file is mapped, all a process has to do to access it is use the data at the address to which the file was mapped. So, using pseudo-code to illustrate the way in which an existing program might be changed to use *mmap*( ), the following:

```
fildes = open(...)
lseek(fildes, some_offset)
read(fildes, buf, len)
/* Use data in buf. */
```

becomes:

```
fildes = open(...)
address = mmap(0, len, PROT_READ, MAP_PRIVATE, fildes, some_offset)
/* Use data at address. */
```

The [EINVAL] error above is marked EX because it is defined as an optional error in the POSIX Realtime Extension.

RATIONALE

After considering several other alternatives, it was decided to adopt the *mmap*( ) definition found in SVR4 for mapping memory objects into process address spaces. The SVR4 definition is minimal, in that it describes only what has been built, and what appears to be necessary for a general and portable mapping facility.

Note that while *mmap*( ) was first designed for mapping files, it is actually a general-purpose mapping facility. It can be used to map any appropriate object, such as memory, files, devices, and so on, into the address space of a process.

When a mapping is established, it is possible that the implementation may need to map more than is requested into the address space of the process because of hardware requirements. An application, however, cannot count on this behavior. Implementations that do not use a paged architecture may simply allocate a common memory region and return the address of it; such implementations probably do not allocate any more than is necessary. References past the end of the requested area are unspecified.

If an application requests a mapping that would overlay existing mappings in the process, it might be desirable that an implementation detect this and inform the application. However, the default, portable (not MAP_FIXED) operation does not overlay existing mappings. On the other hand, if the program specifies a fixed address mapping (which requires some implementation knowledge to determine a suitable address, if the function is supported at all), then the program is presumed to be successfully managing its own address space and should be trusted when it asks to map over existing data structures. Furthermore, it is also desirable to make as few system calls as possible, and it might be considered onerous to require an *munmap*( ) before an *mmap*( ) to the same address range. This volume of IEEE Std 1003.1-200x specifies that the new mappings replace any existing mappings, following existing practice in this regard.

It is not expected, when the Memory Protection option is supported, that all hardware implementations are able to support all combinations of permissions at all addresses. When this option is supported, implementations are required to disallow write access to mappings without write permission and to disallow access to mappings without any access permission. Other than these restrictions, implementations may allow access types other than those requested by the application. For example, if the application requests only PROT_WRITE, the implementation may also allow read access. A call to *mmap*( ) fails if the implementation cannot support allowing

all the access requested by the application. For example, some implementations cannot support a request for both write access and execute access simultaneously. All implementations supporting the Memory Protection option must support requests for no access, read access, write access, and both read and write access. Strictly conforming code must only rely on the required checks. These restrictions allow for portability across a wide range of hardware.

The MAP_FIXED address treatment is likely to fail for non-page-aligned values and for certain architecture-dependent address ranges. Conforming implementations cannot count on being able to choose address values for MAP_FIXED without utilizing non-portable, implementation-defined knowledge. Nonetheless, MAP_FIXED is provided as a standard interface conforming to existing practice for utilizing such knowledge when it is available.

Similarly, in order to allow implementations that do not support virtual addresses, support for directly specifying any mapping addresses via MAP_FIXED is not required and thus a conforming application may not count on it.

The MAP_PRIVATE function can be implemented efficiently when memory protection hardware is available. When such hardware is not available, implementations can implement such ''mappings'' by simply making a real copy of the relevant data into process private memory, though this tends to behave similarly to *read*( ).

The function has been defined to allow for many different models of using shared memory. However, all uses are not equally portable across all machine architectures. In particular, the *mmap*( ) function allows the system as well as the application to specify the address at which to map a specific region of a memory object. The most portable way to use the function is always to let the system choose the address, specifying NULL as the value for the argument *addr* and not to specify MAP_FIXED.

If it is intended that a particular region of a memory object be mapped at the same address in a group of processes (on machines where this is even possible), then MAP_FIXED can be used to pass in the desired mapping address. The system can still be used to choose the desired address if the first such mapping is made without specifying MAP_FIXED, and then the resulting mapping address can be passed to subsequent processes for them to pass in via MAP_FIXED. The availability of a specific address range cannot be guaranteed, in general.

The *mmap*( ) function can be used to map a region of memory that is larger than the current size of the object. Memory access within the mapping but beyond the current end of the underlying objects may result in SIGBUS signals being sent to the process. The reason for this is that the size of the object can be manipulated by other processes and can change at any moment. The implementation should tell the application that a memory reference is outside the object where this can be detected; otherwise, written data may be lost and read data may not reflect actual data in the object.

Note that references beyond the end of the object do not extend the object as the new end cannot be determined precisely by most virtual memory hardware. Instead, the size can be directly manipulated by *ftruncate*( ).

Process memory locking does apply to shared memory regions, and the MEMLOCK_FUTURE argument to *memlockall*( ) can be relied upon to cause new shared memory regions to be automatically locked.

Existing implementations of *mmap*( ) return the value –1 when unsuccessful. Since the casting of this value to type **void *** cannot be guaranteed by the ISO C standard to be distinct from a successful value, this volume of IEEE Std 1003.1-200x defines the symbol MAP_FAILED, which a conforming implementation does not return as the result of a successful call.

25560 **FUTURE DIRECTIONS**

25561 None.

25562 **SEE ALSO**

25563 *exec*, *fcntl*( ), *fork*( ), *lockf*( ), *msync*( ), *munmap*( ), *mprotect*( ), *posix_typed_mem_open*( ), *shmat*( ),
25564 *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

25565 **CHANGE HISTORY**

25566 First released in Issue 4, Version 2.

25567 **Issue 5**

25568 Moved from X/OPEN UNIX extension to BASE.

25569 Aligned with *mmap*( ) in the POSIX Realtime Extension as follows:

25570 • The DESCRIPTION is extensively reworded.

25571 • The [EAGAIN] and [ENOTSUP] mandatory error conditions are added.

25572 • New cases of [ENOMEM] and [ENXIO] are added as mandatory error conditions.

25573 • The value returned on failure is the value of the constant MAP_FAILED; this was previously
25574 defined as –1.

25575 Large File Summit extensions are added.

25576 **Issue 6**

25577 The *mmap*( ) function is marked as part of the Memory Mapped Files option.

25578 The Open Group Corrigendum U028/6 is applied, changing (**void** *)–1 to MAP_FAILED.

25579 The following new requirements on POSIX implementations derive from alignment with the
25580 Single UNIX Specification:

25581 • The DESCRIPTION is updated to described the use of MAP_FIXED.

25582 • The DESCRIPTION is updated to describe the addition of an extra reference to the file
25583 associated with the file descriptor passed to *mmap*( ).

25584 • The DESCRIPTION is updated to state that there may be implementation-defined limits on
25585 the number of memory regions that can be mapped.

25586 • The DESCRIPTION is updated to describe constraints on the alignment and size of the *off*
25587 argument.

25588 • The [EINVAL] and [EMFILE] error conditions are added.

25589 • The [EOVERFLOW] error condition is added. This change is to support large files.

25590 The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

25591 • The DESCRIPTION is updated to describe the cases when MAP_PRIVATE and MAP_FIXED
25592 need not be supported.

25593 The following changes are made for alignment with IEEE Std 1003.1j-2000:

25594 • Semantics for typed memory objects are added to the DESCRIPTION.

25595 • New [ENOMEM] and [ENXIO] errors are added to the ERRORS section.

25596 • The *posix_typed_mem_open*( ) function is added to the SEE ALSO section.

25597 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

25598 **NAME**

25599 modf, modff, modfl — decompose a floating-point number

25600 **SYNOPSIS**

25601 ```
#include <math.h>

double modf(double x, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
```

25605 **DESCRIPTION**

25606 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25607 conflict between the requirements described here and the ISO C standard is unintentional. This
25608 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25609 These functions shall break the argument *x* into integral and fractional parts, each of which has
25610 the same sign as the argument. It stores the integral part as a **double** (for the *modf*( ) function), a
25611 **float** (for the *modff*( ) function), or a **long double** (for the *modfl*( ) function), in the object pointed
25612 to by *iptr*.

25613 **RETURN VALUE**

25614 Upon successful completion, these functions shall return the signed fractional part of *x*.

25615 MX If *x* is NaN, a NaN shall be returned, and **iptr* shall be set to a NaN.

25616 If *x* is ±Inf, ±0 shall be returned, and **iptr* shall be set to ±Inf.

25617 **ERRORS**

25618 No errors are defined.

25619 **EXAMPLES**

25620 None.

25621 **APPLICATION USAGE**

25622 The *modf*( ) function computes the function result and **iptr* such that:

25623 ```
a = modf(x, &iptr) ;
x == a+*iptr ;
```

25625 allowing for the usual floating-point inaccuracies.

25626 **RATIONALE**

25627 None.

25628 **FUTURE DIRECTIONS**

25629 None.

25630 **SEE ALSO**

25631 *frexp*( ), *isnan*( ), *ldexp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

25632 **CHANGE HISTORY**

25633 First released in Issue 1. Derived from Issue 1 of the SVID.

25634 **Issue 5**

25635 The DESCRIPTION is updated to indicate how an application should check for an error. This
25636 text was previously published in the APPLICATION USAGE section.

25637 **Issue 6**

25638 The *modff*( ) and *modfl*( ) functions are added for alignment with the ISO/IEC 9899:1999
25639 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

25644 **NAME**

25645 mprotect — set protection of memory mapping

25646 **SYNOPSIS**

25647 MPR
```
#include <sys/mman.h>
```

25648
```
int mprotect(void *addr, size_t len, int prot);
```

25649

25650 **DESCRIPTION**

25651 The *mprotect*( ) function shall change the access protections to be that specified by *prot* for those
25652 whole pages containing any part of the address space of the process starting at address *addr* and
25653 continuing for *len* bytes. The parameter *prot* determines whether read, write, execute, or some
25654 combination of accesses are permitted to the data being mapped. The *prot* argument should be
25655 either PROT_NONE or the bitwise-inclusive OR of one or more of PROT_READ, PROT_WRITE,
25656 and PROT_EXEC.

25657 If an implementation cannot support the combination of access types specified by *prot*, the call
25658 to *mprotect*( ) shall fail.

25659 An implementation may permit accesses other than those specified by *prot*; however, no
25660 implementation shall permit a write to succeed where PROT_WRITE has not been set or shall
25661 permit any access where PROT_NONE alone has been set. Implementations shall support at
25662 least the following values of *prot*: PROT_NONE, PROT_READ, PROT_WRITE, and the bitwise-
25663 inclusive OR of PROT_READ and PROT_WRITE. If PROT_WRITE is specified, the application
25664 shall ensure that it has opened the mapped objects in the specified address range with write
25665 permission, unless MAP_PRIVATE was specified in the original mapping, regardless of whether
25666 the file descriptors used to map the objects have since been closed.

25667 The implementation shall require that *addr* be a multiple of the page size as returned by
25668 *sysconf*( ).

25669 The behavior of this function is unspecified if the mapping was not established by a call to
25670 *mmap*( ).

25671 When *mprotect*( ) fails for reasons other than [EINVAL], the protections on some of the pages in
25672 the range [*addr*,*addr*+*len*) may have been changed.

25673 **RETURN VALUE**

25674 Upon successful completion, *mprotect*( ) shall return 0; otherwise, it shall return –1 and set *errno*
25675 to indicate the error.

25676 **ERRORS**

25677 The *mprotect*( ) function shall fail if:

25678 [EACCES] The *prot* argument specifies a protection that violates the access permission
25679 the process has to the underlying memory object.

25680 [EAGAIN] The *prot* argument specifies PROT_WRITE over a MAP_PRIVATE mapping
25681 and there are insufficient memory resources to reserve for locking the private
25682 page.

25683 [EINVAL] The *addr* argument is not a multiple of the page size as returned by *sysconf*( ).

25684 [ENOMEM] Addresses in the range [*addr*,*addr*+*len*) are invalid for the address space of a
25685 process, or specify one or more pages which are not mapped.

25686 [ENOMEM] The *prot* argument specifies PROT_WRITE on a MAP_PRIVATE mapping, and
25687 it would require more space than the system is able to supply for locking the
25688 private pages, if required.

[ENOTSUP] The implementation does not support the combination of accesses requested in the *prot* argument.

EXAMPLES

None.

APPLICATION USAGE

The [EINVAL] error above is marked EX because it is defined as an optional error in the POSIX Realtime Extension.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mmap( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Aligned with *mprotect*( ) in the POSIX Realtime Extension as follows:

- The DESCRIPTION is largely reworded.
- [ENOTSUP] and a second form of [ENOMEM] are added as mandatory error conditions.
- [EAGAIN] is moved from the optional to the mandatory error conditions.

Issue 6

The *mprotect*( ) function is marked as part of the Memory Protection option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of the page size as returned by *sysconf*( ).
- The [EINVAL] error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

25718 **NAME**

25719 mq_close — close a message queue (**REALTIME**)

25720 **SYNOPSIS**

25721 MSG
```
#include <mqueue.h>
```

25722
```
int mq_close(mqd_t mqdes);
```

25723

25724 **DESCRIPTION**

25725 The *mq_close*( ) function shall remove the association between the message queue descriptor,
25726 *mqdes*, and its message queue. The results of using this message queue descriptor after
25727 successful return from this *mq_close*( ), and until the return of this message queue descriptor
25728 from a subsequent *mq_open*( ), are undefined.

25729 If the process has successfully attached a notification request to the message queue via this
25730 *mqdes*, this attachment shall be removed, and the message queue is available for another process
25731 to attach for notification.

25732 **RETURN VALUE**

25733 Upon successful completion, the *mq_close*( ) function shall return a value of zero; otherwise, the
25734 function shall return a value of –1 and set *errno* to indicate the error.

25735 **ERRORS**

25736 The *mq_close*( ) function shall fail if:

25737 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

25738 **EXAMPLES**

25739 None.

25740 **APPLICATION USAGE**

25741 None.

25742 **RATIONALE**

25743 None.

25744 **FUTURE DIRECTIONS**

25745 None.

25746 **SEE ALSO**

25747 *mq_open*( ), *mq_unlink*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of
25748 IEEE Std 1003.1-200x, **<mqueue.h>**

25749 **CHANGE HISTORY**

25750 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25751 **Issue 6**

25752 The *mq_close*( ) function is marked as part of the Message Passing option.

25753 The [ENOSYS] error condition has been removed as stubs need not be provided if an
25754 implementation does not support the Message Passing option.

25755 **NAME**

25756 mq_getattr — get message queue attributes (**REALTIME**)

25757 **SYNOPSIS**

25758 MSG
```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
```
25759

25760

25761 **DESCRIPTION**

25762 The *mqdes* argument specifies a message queue descriptor.

25763 The *mq_getattr*( ) function shall obtain status information and attributes of the message queue | 25764 and the open message queue description associated with the message queue descriptor. |

25765 The results shall be returned in the **mq_attr** structure referenced by the *mqstat* argument. |

25766 Upon return, the following members shall have the values associated with the open message | 25767 queue description as set when the message queue was opened and as modified by subsequent | 25768 *mq_setattr*( ) calls: *mq_flags*.

25769 The following attributes of the message queue shall be returned as set at message queue 25770 creation: *mq_maxmsg*, *mq_msgsize*.

25771 Upon return, the following members within the **mq_attr** structure referenced by the *mqstat* | 25772 argument shall be set to the current state of the message queue: |

25773 *mq_curmsgs* The number of messages currently on the queue.

25774 **RETURN VALUE**

25775 Upon successful completion, the *mq_getattr*( ) function shall return zero. Otherwise, the function 25776 shall return –1 and set *errno* to indicate the error.

25777 **ERRORS**

25778 The *mq_getattr*( ) function shall fail if:

25779 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

25780 **EXAMPLES**

25781 None.

25782 **APPLICATION USAGE**

25783 None.

25784 **RATIONALE**

25785 None.

25786 **FUTURE DIRECTIONS**

25787 None.

25788 **SEE ALSO**

25789 *mq_open*( ), *mq_send*( ), *mq_setattr*( ), *mq_timedsend*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the 25790 Base Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**

25791 **CHANGE HISTORY**

25792 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25793 **Issue 6**

25794 The *mq_getattr*( ) function is marked as part of the Message Passing option.

25795 The [ENOSYS] error condition has been removed as stubs need not be provided if an 25796 implementation does not support the Message Passing option.

25797 The *mq_timedsend*( ) function is added to the SEE ALSO section for alignment with
25798 IEEE Std 1003.1d-1999.

25799 **NAME**

25800 mq_notify — notify process that a message is available (**REALTIME**)

25801 **SYNOPSIS**

25802 MSG
```
#include <mqueue.h>

int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```
25803

25804

25805 **DESCRIPTION**

25806 If the argument *notification* is not NULL, this function shall register the calling process to be
25807 notified of message arrival at an empty message queue associated with the specified message
25808 queue descriptor, *mqdes*. The notification specified by the *notification* argument shall be sent to
25809 the process when the message queue transitions from empty to non-empty. At any time, only
25810 one process may be registered for notification by a message queue. If the calling process or any
25811 other process has already registered for notification of message arrival at the specified message
25812 queue, subsequent attempts to register for that message queue shall fail.

25813 If *notification* is NULL and the process is currently registered for notification by the specified
25814 message queue, the existing registration shall be removed.

25815 When the notification is sent to the registered process, its registration shall be removed. The
25816 message queue shall then be available for registration.

25817 If a process has registered for notification of message arrival at a message queue and some
25818 thread is blocked in *mq_receive*( ) waiting to receive a message when a message arrives at the
25819 queue, the arriving message shall satisfy the appropriate *mq_receive*( ). The resulting behavior is
25820 as if the message queue remains empty, and no notification shall be sent.

25821 **RETURN VALUE**

25822 Upon successful completion, the *mq_notify*( ) function shall return a value of zero; otherwise, the
25823 function shall return a value of –1 and set *errno* to indicate the error.

25824 **ERRORS**

25825 The *mq_notify*( ) function shall fail if:

25826 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

25827 [EBUSY] A process is already registered for notification by the message queue.

25828 **EXAMPLES**

25829 None.

25830 **APPLICATION USAGE**

25831 None.

25832 **RATIONALE**

25833 None.

25834 **FUTURE DIRECTIONS**

25835 None.

25836 **SEE ALSO**

25837 *mq_open*( ), *mq_send*( ), *mq_timedsend*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base
25838 Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**

25839 **CHANGE HISTORY**

25840 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25841 **Issue 6**

25842 The *mq_notify*( ) function is marked as part of the Message Passing option.

25843 The [ENOSYS] error condition has been removed as stubs need not be provided if an
25844 implementation does not support the Message Passing option.

25845 The *mq_timedsend*( ) function is added to the SEE ALSO section for alignment with
25846 IEEE Std 1003.1d-1999.

25847 **NAME**

25848 mq_open — open a message queue (**REALTIME**)

25849 **SYNOPSIS**

25850 MSG
```
#include <mqueue.h>
```

25851
```
mqd_t mq_open(const char *name, int oflag, ...);
```

25852

25853 **DESCRIPTION**

25854 The *mq_open*( ) function shall establish the connection between a process and a message queue
25855 with a message queue descriptor. It shall create an open message queue description that refers to
25856 the message queue, and a message queue descriptor that refers to that open message queue
25857 description. The message queue descriptor is used by other functions to refer to that message
25858 queue. The *name* argument points to a string naming a message queue. It is unspecified whether
25859 the name appears in the file system and is visible to other functions that take pathnames as |
25860 arguments. The *name* argument shall conform to the construction rules for a pathname. If *name* |
25861 begins with the slash character, then processes calling *mq_open*( ) with the same value of *name* |
25862 shall refer to the same message queue object, as long as that name has not been removed. If *name* |
25863 does not begin with the slash character, the effect is implementation-defined. The interpretation
25864 of slash characters other than the leading slash character in *name* is implementation-defined. If
25865 the *name* argument is not the name of an existing message queue and creation is not requested,
25866 *mq_open*( ) shall fail and return an error.

25867 A message queue descriptor may be implemented using a file descriptor, in which case
25868 applications can open up to at least {OPEN_MAX} file and message queues.

25869 The *oflag* argument requests the desired receive and/or send access to the message queue. The
25870 requested access permission to receive messages or send messages shall be granted if the calling |
25871 process would be granted read or write access, respectively, to an equivalently protected file. |

25872 The value of *oflag* is the bitwise-inclusive OR of values from the following list. Applications |
25873 shall specify exactly one of the first three values (access modes) below in the value of *oflag*: |

25874 O_RDONLY Open the message queue for receiving messages. The process can use the
25875 returned message queue descriptor with *mq_receive*( ), but not *mq_send*( ). A
25876 message queue may be open multiple times in the same or different processes
25877 for receiving messages.

25878 O_WRONLY Open the queue for sending messages. The process can use the returned
25879 message queue descriptor with *mq_send*( ) but not *mq_receive*( ). A message
25880 queue may be open multiple times in the same or different processes for
25881 sending messages.

25882 O_RDWR Open the queue for both receiving and sending messages. The process can use
25883 any of the functions allowed for O_RDONLY and O_WRONLY. A message
25884 queue may be open multiple times in the same or different processes for
25885 sending messages.

25886 Any combination of the remaining flags may be specified in the value of *oflag*:

25887 O_CREAT Create a message queue. It requires two additional arguments: *mode*, which |
25888 shall be of type **mode_t**, and *attr*, which shall be a pointer to a **mq_attr** |
25889 structure. If the pathname *name* has already been used to create a message |
25890 queue that still exists, then this flag shall have no effect, except as noted under |
25891 O_EXCL. Otherwise, a message queue shall be created without any messages |
25892 in it. The user ID of the message queue shall be set to the effective user ID of |
25893 the process, and the group ID of the message queue shall be set to the effective |

group ID of the process. The file permission bits shall be set to the value of *mode*. When bits in *mode* other than file permission bits are set, the effect is implementation-defined. If *attr* is NULL, the message queue shall be created with implementation-defined default message queue attributes. If *attr* is non-NULL and the calling process has the appropriate privilege on *name*, the message queue *mq_maxmsg* and *mq_msgsize* attributes shall be set to the values of the corresponding members in the **mq_attr** structure referred to by *attr*. If *attr* is non-NULL, but the calling process does not have the appropriate privilege on *name*, the *mq_open*( ) function shall fail and return an error without creating the message queue.

O_EXCL — If O_EXCL and O_CREAT are set, *mq_open*( ) shall fail if the message queue *name* exists. The check for the existence of the message queue and the creation of the message queue if it does not exist shall be atomic with respect to other threads executing *mq_open*( ) naming the same *name* with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the result is undefined.

O_NONBLOCK — Determines whether a *mq_send*( ) or *mq_receive*( ) waits for resources or messages that are not currently available, or fails with *errno* set to [EAGAIN]; see *mq_send*( ) and *mq_receive*( ) for details.

The *mq_open*( ) function does not add or remove messages from the queue.

RETURN VALUE

Upon successful completion, the function shall return a message queue descriptor; otherwise, the function shall return (**mqd_t**)–1 and set *errno* to indicate the error.

ERRORS

The *mq_open*( ) function shall fail if:

[EACCES] The message queue exists and the permissions specified by *oflag* are denied, or the message queue does not exist and permission to create the message queue is denied.

[EEXIST] O_CREAT and O_EXCL are set and the named message queue already exists.

[EINTR] The *mq_open*( ) function was interrupted by a signal.

[EINVAL] The *mq_open*( ) function is not supported for the given name.

[EINVAL] O_CREAT was specified in *oflag*, the value of *attr* is not NULL, and either *mq_maxmsg* or *mq_msgsize* was less than or equal to zero.

[EMFILE] Too many message queue descriptors or file descriptors are currently in use by this process.

[ENAMETOOLONG] The length of the *name* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

[ENFILE] Too many message queues are currently open in the system.

[ENOENT] O_CREAT is not set and the named message queue does not exist.

[ENOSPC] There is insufficient space for the creation of the new message queue.

25935 **EXAMPLES**

25936 None.

25937 **APPLICATION USAGE**

25938 None.

25939 **RATIONALE**

25940 None.

25941 **FUTURE DIRECTIONS**

25942 None.

25943 **SEE ALSO**

25944 *mq_close*( ), *mq_getattr*( ), *mq_receive*( ), *mq_send*( ), *mq_setattr*( ), *mq_timedreceive*( ), *mq_timedsend*( ),
25945 *mq_unlink*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of
25946 IEEE Std 1003.1-200x, **<mqueue.h>**

25947 **CHANGE HISTORY**

25948 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25949 **Issue 6**

25950 The *mq_open*( ) function is marked as part of the Message Passing option.

25951 The [ENOSYS] error condition has been removed as stubs need not be provided if an
25952 implementation does not support the Message Passing option.

25953 The *mq_timedreceive*( ) and *mq_timedsend*( ) functions are added to the SEE ALSO section for
25954 alignment with IEEE Std 1003.1d-1999.

25955 The DESCRIPTION of O_EXCL is updated in response to IEEE PASC Interpretation 1003.1c #48.

NAME

mq_receive, mq_timedreceive — receive a message from a message queue (**REALTIME**)

SYNOPSIS

MSG
```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
    unsigned *msg_prio);
```

MSG TMO
```
#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,
    size_t msg_len, unsigned *restrict msg_prio,
    const struct timespec *restrict abs_timeout);
```

DESCRIPTION

The *mq_receive*( ) function shall receive the oldest of the highest priority message(s) from the message queue specified by *mqdes*. If the size of the buffer in bytes, specified by the *msg_len* argument, is less than the *mq_msgsize* attribute of the message queue, the function shall fail and return an error. Otherwise, the selected message shall be removed from the queue and copied to the buffer pointed to by the *msg_ptr* argument.

If the value of *msg_len* is greater than {SSIZE_MAX}, the result is implementation-defined.

If the argument *msg_prio* is not NULL, the priority of the selected message shall be stored in the location referenced by *msg_prio*.

If the specified message queue is empty and O_NONBLOCK is not set in the message queue description associated with *mqdes*, *mq_receive*( ) shall block until a message is enqueued on the message queue or until *mq_receive*( ) is interrupted by a signal. If more than one thread is waiting to receive a message when a message arrives at an empty queue and the Priority Scheduling option is supported, then the thread of highest priority that has been waiting the longest shall be selected to receive the message. Otherwise, it is unspecified which waiting thread receives the message. If the specified message queue is empty and O_NONBLOCK is set in the message queue description associated with *mqdes*, no message shall be removed from the queue, and *mq_receive*( ) shall return an error.

TMO The *mq_timedreceive*( ) function shall receive the oldest of the highest priority messages from the message queue specified by *mqdes* as described for the *mq_receive*( ) function. However, if O_NONBLOCK was not specified when the message queue was opened via the *mq_open*( ) function, and no message exists on the queue to satisfy the receive, the wait for such a message shall be terminated when the specified timeout expires. If O_NONBLOCK is set, this function is equivalent to *mq_receive*( ).

The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

TMO TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout shall be based on the system clock as returned by the *time*( ) function.

TMO The resolution of the timeout shall be the resolution of the clock on which it is based. The *timespec* argument is defined in the **<time.h>** header.

Under no circumstance shall the operation fail with a timeout if a message can be removed from the message queue immediately. The validity of the *abs_timeout* parameter need not be checked if a message can be removed from the message queue immediately.

RETURN VALUE

TMO Upon successful completion, the *mq_receive*( ) and *mq_timedreceive*( ) functions shall return the length of the selected message in bytes and the message shall be removed from the queue. Otherwise, no message shall be removed from the queue, the functions shall return a value of –1, and set *errno* to indicate the error.

ERRORS

TMO The *mq_receive*( ) and *mq_timedreceive*( ) functions shall fail if:

[EAGAIN] O_NONBLOCK was set in the message description associated with *mqdes*, and the specified message queue is empty.

[EBADF] The *mqdes* argument is not a valid message queue descriptor open for reading.

[EMSGSIZE] The specified message buffer size, *msg_len*, is less than the message size attribute of the message queue.

TMO [EINTR] The *mq_receive*( ) or *mq_timedreceive*( ) operation was interrupted by a signal.

TMO [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1 000 million.

TMO [ETIMEDOUT] The O_NONBLOCK flag was not set when the message queue was opened, but no message arrived on the queue before the specified timeout expired.

TMO The *mq_receive*( ) and *mq_timedreceive*( ) functions may fail if:

[EBADMSG] The implementation has detected a data corruption problem with the message.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_open( ), *mq_send*( ), *mq_timedsend*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_receive*( ) function is marked as part of the Message Passing option.

The Open Group Corrigendum U021/4 is applied. The DESCRIPTION is changed to refer to *msg_len* rather than *maxsize*.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In this function it is possible for the return value to exceed the range of the type **ssize_t** (since **size_t** has a larger range of positive values than **ssize_t**). A sentence restricting the size of the **size_t** object is added to the description to resolve this conflict.

The *mq_timedreceive*( ) function is added for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *mq_timedreceive*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

IEEE PASC Interpretation 1003.1 #109 is applied, correcting the return type for *mq_timedreceive*( ) from **int** to **ssize_t**.

NAME

mq_send, mq_timedsend — send a message to a message queue (**REALTIME**)

SYNOPSIS

MSG
```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
    unsigned msg_prio);
```

MSG TMO
```
#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
    unsigned msg_prio, const struct timespec *abs_timeout);
```

DESCRIPTION

The *mq_send*( ) function shall add the message pointed to by the argument *msg_ptr* to the message queue specified by *mqdes*. The *msg_len* argument specifies the length of the message, in bytes, pointed to by *msg_ptr*. The value of *msg_len* shall be less than or equal to the *mq_msgsize* attribute of the message queue, or *mq_send*( ) shall fail.

If the specified message queue is not full, *mq_send*( ) shall behave as if the message is inserted into the message queue at the position indicated by the *msg_prio* argument. A message with a larger numeric value of *msg_prio* shall be inserted before messages with lower values of *msg_prio*. A message shall be inserted after other messages in the queue, if any, with equal *msg_prio*. The value of *msg_prio* shall be less than {MQ_PRIO_MAX}.

If the specified message queue is full and O_NONBLOCK is not set in the message queue description associated with *mqdes*, *mq_send*( ) shall block until space becomes available to enqueue the message, or until *mq_send*( ) is interrupted by a signal. If more than one thread is waiting to send when space becomes available in the message queue and the Priority Scheduling option is supported, then the thread of the highest priority that has been waiting the longest shall be unblocked to send its message. Otherwise, it is unspecified which waiting thread is unblocked. If the specified message queue is full and O_NONBLOCK is set in the message queue description associated with *mqdes*, the message shall not be queued and *mq_send*( ) shall return an error.

TMO The *mq_timedsend*( ) function shall add a message to the message queue specified by *mqdes* in the manner defined for the *mq_send*( ) function. However, if the specified message queue is full and O_NONBLOCK is not set in the message queue description associated with *mqdes*, the wait for sufficient room in the queue shall be terminated when the specified timeout expires. If O_NONBLOCK is set in the message queue description, this function shall be equivalent to *mq_send*( ).

The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

TMO TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout shall be based on the system clock as returned by the *time*( ) function.

TMO The resolution of the timeout shall be the resolution of the clock on which it is based. The *timespec* argument is defined in the **<time.h>** header.

Under no circumstance shall the operation fail with a timeout if there is sufficient room in the queue to add the message immediately. The validity of the *abs_timeout* parameter need not be checked when there is sufficient room in the queue.

RETURN VALUE

TMO Upon successful completion, the *mq_send*( ) and *mq_timedsend*( ) functions shall return a value of zero. Otherwise, no message shall be enqueued, the functions shall return –1, and *errno* shall be set to indicate the error.

ERRORS

TMO The *mq_send*( ) and *mq_timedsend*( ) functions shall fail if:

[EAGAIN] The O_NONBLOCK flag is set in the message queue description associated with *mqdes*, and the specified message queue is full.

[EBADF] The *mqdes* argument is not a valid message queue descriptor open for writing.

TMO [EINTR] A signal interrupted the call to *mq_send*( ) or *mq_timedsend*( ).

[EINVAL] The value of *msg_prio* was outside the valid range.

TMO [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1 000 million.

[EMSGSIZE] The specified message length, *msg_len*, exceeds the message size attribute of the message queue.

TMO [ETIMEDOUT] The O_NONBLOCK flag was not set when the message queue was opened, but the timeout expired before the message could be added to the queue.

EXAMPLES

None.

APPLICATION USAGE

The value of the symbol {MQ_PRIO_MAX} limits the number of priority levels supported by the application. Message priorities range from 0 to {MQ_PRIO_MAX}–1.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_open( ), *mq_receive*( ), *mq_setattr*( ), *mq_timedreceive*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_send*( ) function is marked as part of the Message Passing option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

The *mq_timedsend*( ) function is added for alignment with IEEE Std 1003.1d-1999.

26141 **NAME**

26142 mq_setattr — set message queue attributes (**REALTIME**)

26143 **SYNOPSIS**

26144 MSG
```
#include <mqueue.h>

int mq_setattr(mqd_t mqdes, const struct mq_attr *restrict mqstat,
    struct mq_attr *restrict omqstat);
```
26147

26148 **DESCRIPTION**

26149 The *mq_setattr*( ) function shall set attributes associated with the open message queue | 26150 description referenced by the message queue descriptor specified by *mqdes*. |

26151 The message queue attributes corresponding to the following members defined in the **mq_attr** | 26152 structure shall be set to the specified values upon successful completion of *mq_setattr*( ): |

26153 *mq_flags* The value of this member is the bitwise-logical OR of zero or more of 26154 O_NONBLOCK and any implementation-defined flags.

26155 The values of the *mq_maxmsg*, *mq_msgsize*, and *mq_curmsgs* members of the **mq_attr** structure | 26156 shall be ignored by *mq_setattr*( ). |

26157 If *omqstat* is non-NULL, the *mq_setattr*( ) function shall store, in the location referenced by | 26158 *omqstat*, the previous message queue attributes and the current queue status. These values shall | 26159 be the same as would be returned by a call to *mq_getattr*( ) at that point. |

26160 **RETURN VALUE**

26161 Upon successful completion, the function shall return a value of zero and the attributes of the 26162 message queue shall have been changed as specified.

26163 Otherwise, the message queue attributes shall be unchanged, and the function shall return a 26164 value of −1 and set *errno* to indicate the error.

26165 **ERRORS**

26166 The *mq_setattr*( ) function shall fail if:

26167 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

26168 **EXAMPLES**

26169 None.

26170 **APPLICATION USAGE**

26171 None.

26172 **RATIONALE**

26173 None.

26174 **FUTURE DIRECTIONS**

26175 None.

26176 **SEE ALSO**

26177 *mq_open*( ), *mq_send*( ), *mq_timedsend*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base 26178 Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**

26179 **CHANGE HISTORY**

26180 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_setattr*( ) function is marked as part of the Message Passing option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

The *mq_timedsend*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *mq_setattr*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

26189 **NAME**

26190 mq_timedreceive — receive a message from a message queue (**ADVANCED REALTIME**)

26191 **SYNOPSIS**

26192 MSG TMO
```
#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,
    size_t msg_len, unsigned *restrict msg_prio,
    const struct timespec *restrict abs_timeout);
```
26193

26194

26195

26196

26197

26198 **DESCRIPTION**

26199 Refer to *mq_receive*( ).

NAME

mq_timedsend — send a message to a message queue (**ADVANCED REALTIME**)

SYNOPSIS

MSG TMO

```
#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
    unsigned msg_prio, const struct timespec *abs_timeout);
```

DESCRIPTION

Refer to *mq_send*( ).

26210 **NAME**

26211 mq_unlink — remove a message queue (**REALTIME**)

26212 **SYNOPSIS**

26213 MSG
```
#include <mqueue.h>

int mq_unlink(const char *name);
```
26214

26215

26216 **DESCRIPTION**

26217 The *mq_unlink*( ) function shall remove the message queue named by the pathname *name*. After |
26218 a successful call to *mq_unlink*( ) with *name*, a call to *mq_open*( ) with *name* shall fail if the flag |
26219 O_CREAT is not set in *flags*. If one or more processes have the message queue open when |
26220 *mq_unlink*( ) is called, destruction of the message queue shall be postponed until all references to |
26221 the message queue have been closed.

26222 Calls to *mq_open*( ) to recreate the message queue may fail until the message queue is actually
26223 removed. However, the *mq_unlink*( ) call need not block until all references have been closed; it
26224 may return immediately.

26225 **RETURN VALUE**

26226 Upon successful completion, the function shall return a value of zero. Otherwise, the named
26227 message queue shall be unchanged by this function call, and the function shall return a value of
26228 –1 and set *errno* to indicate the error.

26229 **ERRORS**

26230 The *mq_unlink*( ) function shall fail if:

26231 [EACCES] Permission is denied to unlink the named message queue.

26232 [ENAMETOOLONG]
26233 The length of the *name* argument exceeds {PATH_MAX} or a pathname |
26234 component is longer than {NAME_MAX}. |

26235 [ENOENT] The named message queue does not exist.

26236 **EXAMPLES**

26237 None.

26238 **APPLICATION USAGE**

26239 None.

26240 **RATIONALE**

26241 None.

26242 **FUTURE DIRECTIONS**

26243 None.

26244 **SEE ALSO**

26245 *mq_close*( ), *mq_open*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of
26246 IEEE Std 1003.1-200x, **<mqueue.h>**

26247 **CHANGE HISTORY**

26248 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

26249 **Issue 6**

26250 The *mq_unlink*( ) function is marked as part of the Message Passing option.

26251 The Open Group Corrigendum U021/5 is applied, clarifying that upon unsuccessful completion,
26252 the named message queue is unchanged by this function.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

NAME

mrand48 — generate uniformly distributed pseudo-random signed long integers

SYNOPSIS

XSI

```
#include <stdlib.h>

long mrand48(void);
```

DESCRIPTION

Refer to *drand48*( ).

NAME

msgctl — XSI message control operations

SYNOPSIS

XSI
```
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

DESCRIPTION

The *msgctl*( ) function operates on XSI message queues (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.224, Message Queue). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *msgctl*( ) function shall provide message control operations as specified by *cmd*. The following values for *cmd*, and the message control operations they specify, are:

IPC_STAT
: Place the current value of each member of the **msqid_ds** data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in **<sys/msg.h>**.

IPC_SET
: Set the value of the following members of the **msqid_ds** data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode
msg_qbytes
```

: IPC_SET can only be executed by a process with appropriate privileges or that has an effective user ID equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the **msqid_ds** data structure associated with *msqid*. Only a process with appropriate privileges can raise the value of **msg_qbytes**.

IPC_RMID
: Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and **msqid_ds** data structure associated with it. IPC_RMD can only be executed by a process with appropriate privileges or one that has an effective user ID equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the **msqid_ds** data structure associated with *msqid*.

RETURN VALUE

Upon successful completion, *msgctl*( ) shall return 0; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *msgctl*( ) function shall fail if:

[EACCES]
: The argument *cmd* is IPC_STAT and the calling process does not have read permission; see Section 2.7 (on page 489).

[EINVAL]
: The value of *msqid* is not a valid message queue identifier; or the value of *cmd* is not a valid command.

[EPERM]
: The argument *cmd* is IPC_RMID or IPC_SET and the effective user ID of the calling process is not equal to that of a process with appropriate privileges and it is not equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.

[EPERM] The argument *cmd* is IPC_SET, an attempt is being made to increase to the value of **msg_qbytes**, and the effective user ID of the calling process does not have appropriate privileges.

EXAMPLES

None.

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_close( ), *mq_getattr*( ), *mq_notify*( ), *mq_open*( ), *mq_receive*( ), *mq_send*( ), *mq_setattr*( ), *mq_unlink*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/msg.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

NAME

msgget — get the XSI message queue identifier

SYNOPSIS

XSI
```
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

DESCRIPTION

The *msgget*( ) function operates on XSI message queues (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.224, Message Queue). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *msgget*( ) function shall return the message queue identifier associated with the argument *key*.

A message queue identifier, associated message queue, and data structure (see **<sys/msg.h>**), shall be created for the argument *key* if one of the following is true:

- The argument *key* is equal to IPC_PRIVATE.
- The argument *key* does not already have a message queue identifier associated with it, and (*msgflg* & IPC_CREAT) is non-zero.

Upon creation, the data structure associated with the new message queue identifier shall be initialized as follows:

- **msg_perm.cuid**, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** shall be set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of **msg_perm.mode** shall be set equal to the low-order 9 bits of *msgflg*.
- **msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** shall be set equal to 0.
- **msg_ctime** shall be set equal to the current time.
- **msg_qbytes** shall be set equal to the system limit.

RETURN VALUE

Upon successful completion, *msgget*( ) shall return a non-negative integer, namely a message queue identifier. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *msgget*( ) function shall fail if:

[EACCES] A message queue identifier exists for the argument *key*, but operation permission as specified by the low-order 9 bits of *msgflg* would not be granted; see Section 2.7 (on page 489).

[EEXIST] A message queue identifier exists for the argument *key* but ((*msgflg* & IPC_CREAT) && (*msgflg* & IPC_EXCL)) is non-zero.

[ENOENT] A message queue identifier does not exist for the argument *key* and (*msgflg* & IPC_CREAT) is 0.

[ENOSPC] A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system-wide would be exceeded.

EXAMPLES

None.

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_close( ), *mq_getattr*( ), *mq_notify*( ), *mq_open*( ), *mq_receive*( ), *mq_send*( ), *mq_setattr*( ), *mq_unlink*( ), *msgctl*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/msg.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

NAME

msgrcv — XSI message receive operation

SYNOPSIS

XSI
```
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
    int msgflg);
```

DESCRIPTION

The *msgrcv*( ) function operates on XSI message queues (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.224, Message Queue). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *msgrcv*( ) function shall read a message from the queue associated with the message queue identifier specified by *msqid* and place it in the user-defined buffer pointed to by *msgp*.

The application shall ensure that the argument *msgp* points to a user-defined buffer that contains first a field of type **long** specifying the type of the message, and then a data portion that holds the data bytes of the message. The structure below is an example of what this user-defined buffer might look like:

```
struct mymsg {
    long    mtype;     /* Message type. */
    char    mtext[1];  /* Message text. */
}
```

The structure member *mtype* is the received message's type as specified by the sending process.

The structure member *mtext* is the text of the message.

The argument *msgsz* specifies the size in bytes of *mtext*. The received message shall be truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & MSG_NOERROR) is non-zero. The truncated part of the message shall be lost and no indication of the truncation shall be given to the calling process.

If the value of *msgsz* is greater than {SSIZE_MAX}, the result is implementation-defined.

The argument *msgtyp* specifies the type of message requested as follows:

- If *msgtyp* is 0, the first message on the queue shall be received.
- If *msgtyp* is greater than 0, the first message of type *msgtyp* shall be received.
- If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* shall be received.

The argument *msgflg* specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

- If (*msgflg* & IPC_NOWAIT) is non-zero, the calling thread shall return immediately with a return value of –1 and *errno* set to [ENOMSG].
- If (*msgflg* & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the following occurs:
  - — A message of the desired type is placed on the queue.
  - — The message queue identifier *msqid* is removed from the system; when this occurs, *errno* shall be set equal to [EIDRM] and –1 shall be returned.

— The calling thread receives a signal that is to be caught; in this case a message is not received and the calling thread resumes execution in the manner prescribed in *sigaction*( ).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*:

- **msg_qnum** shall be decremented by 1.
- **msg_lrpid** shall be set equal to the process ID of the calling process.
- **msg_rtime** shall be set equal to the current time.

RETURN VALUE

Upon successful completion, *msgrcv*( ) shall return a value equal to the number of bytes actually placed into the buffer *mtext*. Otherwise, no message shall be received, *msgrcv*( ) shall return (**ssize_t**)–1, and *errno* shall be set to indicate the error.

ERRORS

The *msgrcv*( ) function shall fail if:

[E2BIG]	The value of *mtext* is greater than *msgsz* and (*msgflg* & MSG_NOERROR) is 0.
[EACCES]	Operation permission is denied to the calling process; see Section 2.7 (on page 489).
[EIDRM]	The message queue identifier *msqid* is removed from the system.
[EINTR]	The *msgrcv*( ) function was interrupted by a signal.
[EINVAL]	*msqid* is not a valid message queue identifier.
[ENOMSG]	The queue does not contain a message of the desired type and (*msgflg* & IPC_NOWAIT) is non-zero.

EXAMPLES

Receiving a Message

The following example receives the first message on the queue (based on the value of the *msgtyp* argument, 0). The queue is identified by the *msqid* argument (assuming that the value has previously been set). This call specifies that an error should be reported if no message is available, but not if the message is too large. The message size is calculated directly using the *sizeof* operator.

```
#include <sys/msg.h>
...
int result;
int msqid;
struct message {
    long type;
    char text[20];
} msg;
long msgtyp = 0;
...
result = msgrcv(msqid, (void *) &msg, sizeof(msg.text),
         msgtyp, MSG_NOERROR | IPC_NOWAIT);
```

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_close( ), *mq_getattr*( ), *mq_notify*( ), *mq_open*( ), *mq_receive*( ), *mq_send*( ), *mq_setattr*( ), *mq_unlink*( ), *msgctl*( ), *msgget*( ), *msgsnd*( ), *sigaction*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/msg.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The type of the return value is changed from **int** to **ssize_t**, and a warning is added to the DESCRIPTION about values of *msgsz* larger the {SSIZE_MAX}.

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to the APPLICATION USAGE section.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

msgsnd — XSI message send operation

SYNOPSIS

XSI

```
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

DESCRIPTION

The *msgsnd*( ) function operates on XSI message queues (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.224, Message Queue). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *msgsnd*( ) function shall send a message to the queue associated with the message queue identifier specified by *msqid*.

The application shall ensure that the argument *msgp* points to a user-defined buffer that contains first a field of type **long** specifying the type of the message, and then a data portion that holds the data bytes of the message. The structure below is an example of what this user-defined buffer might look like:

```
struct mymsg {
    long   mtype;       /* Message type. */
    char   mtext[1];    /* Message text. */
}
```

The structure member *mtype* is a non-zero positive type **long** that can be used by the receiving process for message selection.

The structure member *mtext* is any text of length *msgsz* bytes. The argument *msgsz* can range from 0 to a system-imposed maximum.

The argument *msgflg* specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to **msg_qbytes**; see **<sys/msg.h>**.
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (*msgflg* & IPC_NOWAIT) is non-zero, the message shall not be sent and the calling thread shall return immediately.
- If (*msgflg* & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the following occurs:
  - — The condition responsible for the suspension no longer exists, in which case the message is sent.
  - — The message queue identifier *msqid* is removed from the system; when this occurs, *errno* shall be set equal to [EIDRM] and –1 shall be returned.
  - — The calling thread receives a signal that is to be caught; in this case the message is not sent and the calling thread resumes execution in the manner prescribed in *sigaction*( ).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*; see **<sys/msg.h>**:

- **msg_qnum** shall be incremented by 1.
- **msg_lspid** shall be set equal to the process ID of the calling process.
- **msg_stime** shall be set equal to the current time.

RETURN VALUE

Upon successful completion, *msgsnd*( ) shall return 0; otherwise, no message shall be sent, *msgsnd*( ) shall return –1, and *errno* shall be set to indicate the error.

ERRORS

The *msgsnd*( ) function shall fail if:

[EACCES] Operation permission is denied to the calling process; see Section 2.7 (on page 489).

[EAGAIN] The message cannot be sent for one of the reasons cited above and (*msgflg* & IPC_NOWAIT) is non-zero.

[EIDRM] The message queue identifier *msgid* is removed from the system.

[EINTR] The *msgsnd*( ) function was interrupted by a signal.

[EINVAL] The value of *msqid* is not a valid message queue identifier, or the value of *mtype* is less than 1; or the value of *msgsz* is less than 0 or greater than the system-imposed limit.

EXAMPLES

Sending a Message

The following example sends a message to the queue identified by the *msqid* argument (assuming that value has previously been set). This call specifies that an error should be reported if no message is available. The message size is calculated directly using the *sizeof* operator.

```
#include <sys/msg.h>
...
int result;
int msqid;
struct message {
    long type;
    char text[20];
} msg;

msg.type = 1;
strcpy(msg.text, "This is message 1");
...
result = msgsnd(msqid, (void *) &msg, sizeof(msg.text), IPC_NOWAIT);
```

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_close( ), *mq_getattr*( ), *mq_notify*( ), *mq_open*( ), *mq_receive*( ), *mq_send*( ), *mq_setattr*( ), *mq_unlink*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *sigaction*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/msg.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

msync — synchronize memory with physical storage

SYNOPSIS

MF SIO
```
#include <sys/mman.h>

int msync(void *addr, size_t len, int flags);
```

DESCRIPTION

The *msync*( ) function shall write all modified data to permanent storage locations, if any, in those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes. If no such storage exists, *msync*( ) need not have any effect. If requested, the *msync*( ) function shall then invalidate cached copies of data.

The implementation shall require that *addr* be a multiple of the page size as returned by *sysconf*( ).

For mappings to files, the *msync*( ) function shall ensure that all write operations are completed as defined for synchronized I/O data integrity completion. It is unspecified whether the implementation also writes out other file attributes. When the *msync*( ) function is called on MAP_PRIVATE mappings, any modified data shall not be written to the underlying object and shall not cause such data to be made visible to other processes. It is unspecified whether data in SHM|TYM MAP_PRIVATE mappings has any permanent storage locations. The effect of *msync*( ) on a shared memory object or a typed memory object is unspecified. The behavior of this function is unspecified if the mapping was not established by a call to *mmap*( ).

The *flags* argument is constructed from the bitwise-inclusive OR of one or more of the following flags defined in the **<sys/mman.h>** header:

Symbolic Constant	Description
MS_ASYNC	Perform asynchronous writes.
MS_SYNC	Perform synchronous writes.
MS_INVALIDATE	Invalidate cached data.

When MS_ASYNC is specified, *msync*( ) shall return immediately once all the write operations are initiated or queued for servicing; when MS_SYNC is specified, *msync*( ) shall not return until all write operations are completed as defined for synchronized I/O data integrity completion. Either MS_ASYNC or MS_SYNC is specified, but not both.

When MS_INVALIDATE is specified, *msync*( ) shall invalidate all cached copies of mapped data that are inconsistent with the permanent storage locations such that subsequent references shall obtain data that was consistent with the permanent storage locations sometime between the call to *msync*( ) and the first subsequent memory reference to the data.

If *msync*( ) causes any write to a file, the file's *st_ctime* and *st_mtime* fields shall be marked for update.

RETURN VALUE

Upon successful completion, *msync*( ) shall return 0; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *msync*( ) function shall fail if:

[EBUSY] Some or all of the addresses in the range starting at *addr* and continuing for *len* bytes are locked, and MS_INVALIDATE is specified.

[EINVAL] The value of *flags* is invalid.

[EINVAL] The value of *addr* is not a multiple of the page size, {PAGESIZE}.

[ENOMEM] The addresses in the range starting at *addr* and continuing for *len* bytes are outside the range allowed for the address space of a process or specify one or more pages that are not mapped.

EXAMPLES

None.

APPLICATION USAGE

The *msync*( ) function is only supported if the Memory Mapped Files option and the Synchronized Input and Output option are supported, and thus need not be available on all implementations.

The *msync*( ) function should be used by programs that require a memory object to be in a known state; for example, in building transaction facilities.

Normal system activity can cause pages to be written to disk. Therefore, there are no guarantees that *msync*( ) is the only control over when pages are or are not written to disk.

RATIONALE

The *msync*( ) function writes out data in a mapped region to the permanent storage for the underlying object. The call to *msync*( ) ensures data integrity of the file.

After the data is written out, any cached data may be invalidated if the MS_INVALIDATE flag was specified. This is useful on systems that do not support read/write consistency.

FUTURE DIRECTIONS

None.

SEE ALSO

mmap( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Aligned with *msync*( ) in the POSIX Realtime Extension as follows:

- The DESCRIPTION is extensively reworded.
- [EBUSY] and a new form of [EINVAL] are added as mandatory error conditions.

Issue 6

The *msync*( ) function is marked as part of the Memory Mapped Files and Synchronized Input and Output options.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [EBUSY] mandatory error condition is added.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of the page size.
- The second [EINVAL] error condition is made mandatory.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding reference to typed memory objects.

26682 **NAME**

26683 munlock — unlock a range of process address space

26684 **SYNOPSIS**

26685 MLR
```
#include <sys/mman.h>
```

26686
```
int munlock(const void *addr, size_t len);
```

26687

26688 **DESCRIPTION**

26689 Refer to *mlock*( ).

26690 **NAME**

26691 munlockall — unlock the address space of a process

26692 **SYNOPSIS**

26693 ML
```
#include <sys/mman.h>
```

26694
```
int munlockall(void);
```

26695

26696 **DESCRIPTION**

26697 Refer to *mlockall*( ).

NAME

munmap — unmap pages of memory

SYNOPSIS

MF|SHM
```
#include <sys/mman.h>

int munmap(void *addr, size_t len);
```

DESCRIPTION

The *munmap*( ) function shall remove any mappings for those entire pages containing any part of the address space of the process starting at *addr* and continuing for *len* bytes. Further references to these pages shall result in the generation of a SIGSEGV signal to the process. If there are no mappings in the specified address range, then *munmap*( ) has no effect.

The implementation shall require that *addr* be a multiple of the page size {PAGESIZE}.

If a mapping to be removed was private, any modifications made in this address range shall be discarded.

ML|MLR Any memory locks (see *mlock*( ) and *mlockall*( )) associated with this address range shall be removed, as if by an appropriate call to *munlock*( ).

TYM If a mapping removed from a typed memory object causes the corresponding address range of the memory pool to be inaccessible by any process in the system except through allocatable mappings (that is, mappings of typed memory objects opened with the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag), then that range of the memory pool shall become deallocated and may become available to satisfy future typed memory allocation requests.

A mapping removed from a typed memory object opened with the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag shall not affect in any way the availability of that typed memory for allocation.

The behavior of this function is unspecified if the mapping was not established by a call to *mmap*( ).

RETURN VALUE

Upon successful completion, *munmap*( ) shall return 0; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *munmap*( ) function shall fail if:

[EINVAL] Addresses in the range [*addr*,*addr*+*len*) are outside the valid range for the address space of a process.

[EINVAL] The *len* argument is 0.

[EINVAL] The *addr* argument is not a multiple of the page size as returned by *sysconf*( ).

EXAMPLES

None.

APPLICATION USAGE

The *munmap*( ) function is only supported if the Memory Mapped Files option or the Shared Memory Objects option is supported.

RATIONALE

The *munmap*( ) function corresponds to SVR4, just as the *mmap*( ) function does.

It is possible that an application has applied process memory locking to a region that contains shared memory. If this has occurred, the *munmap*( ) call ignores those locks and, if necessary, causes those locks to be removed.

FUTURE DIRECTIONS

None.

SEE ALSO

mlock( ), *mlockall*( ), *mmap*( ), *posix_typed_mem_open*( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Aligned with *munmap*( ) in the POSIX Realtime Extension as follows:

- The DESCRIPTION is extensively reworded.
- The SIGBUS error is no longer permitted to be generated.

Issue 6

The *munmap*( ) function is marked as part of the Memory Mapped Files and Shared Memory Objects option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of the page size.
- The [EINVAL] error conditions are added.

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- Semantics for typed memory objects are added to the DESCRIPTION.
- The *posix_typed_mem_open*( ) function is added to the SEE ALSO section.

NAME

nan, nanf, nanl — return quiet NaN

SYNOPSIS

```
#include <math.h>

double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The function call *nan*("*n-char-sequence*") shall be equivalent to:

```
strtod("NAN(n-char-sequence)", (char **) NULL);
```

The function call *nan*("") shall be equivalent to:

```
strtod("NAN()", (char **) NULL)
```

If *tagp* does not point to an *n*-**char** sequence or an empty string, the function call shall be equivalent to:

```
strtod("NAN", (char **) NULL)
```

Function calls to *nanf*() and *nanl*() are equivalent to the corresponding function calls to *strtof*() and *strtold*().

RETURN VALUE

These functions shall return a quiet NaN, if available, with content indicated through *tagp*.

If the implementation does not support quiet NaNs, these functions shall return zero.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strtod(), *strtold*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

nanosleep — high resolution sleep (**REALTIME**)

SYNOPSIS

TMR
```
#include <time.h>

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

DESCRIPTION

The *nanosleep*( ) function shall cause the current thread to be suspended from execution until either the time interval specified by the *rqtp* argument has elapsed or a signal is delivered to the calling thread, and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time shall not be less than the time specified by *rqtp*, as measured by the system clock, CLOCK_REALTIME.

The use of the *nanosleep*( ) function has no effect on the action or blockage of any signal.

RETURN VALUE

If the *nanosleep*( ) function returns because the requested time has elapsed, its return value shall be zero.

If the *nanosleep*( ) function returns because it has been interrupted by a signal, it shall return a value of –1 and set *errno* to indicate the interruption. If the *rmtp* argument is non-NULL, the **timespec** structure referenced by it is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the *rmtp* argument is NULL, the remaining time is not returned.

If *nanosleep*( ) fails, it shall return a value of –1 and set *errno* to indicate the error.

ERRORS

The *nanosleep*( ) function shall fail if:

[EINTR] The *nanosleep*( ) function was interrupted by a signal.

[EINVAL] The *rqtp* argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

It is common to suspend execution of a process for an interval in order to poll the status of a non-interrupting function. A large number of actual needs can be met with a simple extension to *sleep*( ) that provides finer resolution.

In the POSIX.1-1990 standard and SVR4, it is possible to implement such a routine, but the frequency of wakeup is limited by the resolution of the *alarm*( ) and *sleep*( ) functions. In 4.3 BSD, it is possible to write such a routine using no static storage and reserving no system facilities. Although it is possible to write a function with similar functionality to *sleep*( ) using the remainder of the timers function, such a function requires the use of signals and the reservation of some signal number. This volume of IEEE Std 1003.1-200x requires that *nanosleep*( ) be non-intrusive of the signals function.

The *nanosleep*( ) function shall return a value of 0 on success and –1 on failure or if interrupted. This latter case is different from *sleep*( ). This was done because the remaining time is returned via an argument structure pointer, *rmtp*, instead of as the return value.

FUTURE DIRECTIONS

None.

SEE ALSO

sleep( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *nanosleep*( ) function is marked as part of the Timers option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

NAME

nearbyint, nearbyintf, nearbyintl — floating-point rounding functions

SYNOPSIS

```
#include <math.h>

double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall round their argument to an integer value in floating-point format, using the current rounding direction and without raising the inexact floating-point exception.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the rounded integer value.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, ±0 shall be returned.

If *x* is ±Inf, *x* shall be returned.

XSI If the correct value would cause overflow, a range error shall occur and *nearbyint*(), *nearbyintf*(), and *nearbyintl*() shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

ERRORS

These functions shall fail if:

XSI

Range Error	The result would cause an overflow.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

26902 **SEE ALSO**

26903 *feclearexcept*( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, |
26904 Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

26905 **CHANGE HISTORY**

26906 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

nextafter, nextafterf, nextafterl, nexttoward, nexttowardf, nexttowardl — next representable floating-point number

SYNOPSIS

```
#include <math.h>

double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *nextafter*( ), *nextafterf*( ), and *nextafterl*( ) functions shall compute the next representable floating-point value following *x* in the direction of *y*. Thus, if *y* is less than *x*, *nextafter*( ) shall return the largest representable floating-point number less than *x*. The *nextafter*( ), *nextafterf*( ), and *nextafterl*( ) functions shall return *y* if *x* equals *y*.

The *nexttoward*( ), *nexttowardf*( ), and *nexttowardl*( ) functions shall be equivalent to the corresponding *nextafter*( ) functions, except that the second parameter shall have type **long double** and the functions shall return *y* converted to the type of the function if *x* equals *y*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the next representable floating-point value following *x* in the direction of *y*.

If *x*==*y*, *y* (of the type *x*) shall be returned.

If *x* is finite and the correct function value would overflow, a range error shall occur and ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (with the same sign as *x*) shall be returned as appropriate for the return type of the function.

MX If *x* or *y* is NaN, a NaN shall be returned.

If *x*!=*y* and the correct function value is subnormal, zero, or underflows, a range error shall occur, and either the correct function value (if representable) or 0.0 shall be returned.

ERRORS

These functions shall fail if:

Range Error	The correct value overflows
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

MX

Range Error	The correct value is subnormal or underflows

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *nextafter*( ) function is no longer marked as an extension.

The *nextafterf*( ), *nextafterl*( ), *nexttoward*( ), *nexttowardf*( ), *nexttowardl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

26979 **NAME**

26980 nexttoward, nexttowardf, nexttowardl — next representable floating-point number

26981 **SYNOPSIS**

26982 `#include <math.h>`

26983 `double nexttoward(double x, long double y);`
26984 `float nexttowardf(float x, long double y);`
26985 `long double nexttowardl(long double x, long double y);`

26986 **DESCRIPTION**

26987 Refer to *nextafter*( ).

NAME

nftw — walk a file tree

SYNOPSIS

XSI

```
#include <ftw.h>

int nftw(const char *path, int (*fn)(const char *,
    const struct stat *, int, struct FTW *), int depth, int flags);
```

DESCRIPTION

The *nftw*() function shall recursively descend the directory hierarchy rooted in *path*. The *nftw*() function has a similar effect to *ftw*() except that it takes an additional argument *flags*, which is a bitwise-inclusive OR of zero or more of the following flags:

FTW_CHDIR — If set, *nftw*() shall change the current working directory to each directory as it reports files in that directory. If clear, *nftw*() shall not change the current working directory.

FTW_DEPTH — If set, *nftw*() shall report all files in a directory before reporting the directory itself. If clear, *nftw*() shall report any directory before reporting the files in that directory.

FTW_MOUNT — If set, *nftw*() shall only report files in the same file system as *path*. If clear, *nftw*() shall report all files encountered during the walk.

FTW_PHYS — If set, *nftw*() shall perform a physical walk and shall not follow symbolic links.

If FTW_PHYS is clear and FTW_DEPTH is set, *nftw*() shall follow links instead of reporting them, but shall not report any directory that would be a descendant of itself. If FTW_PHYS is clear and FTW_DEPTH is clear, *nftw*() shall follow links instead of reporting them, but shall not report the contents of any directory that would be a descendant of itself.

At each file it encounters, *nftw*() shall call the user-supplied function *fn* with four arguments:

- The first argument is the pathname of the object.
- The second argument is a pointer to the **stat** buffer containing information on the object.
- The third argument is an integer giving additional information. Its value is one of the following:

  FTW_F — The object is a file.

  FTW_D — The object is a directory.

  FTW_DP — The object is a directory and subdirectories have been visited. (This condition shall only occur if the FTW_DEPTH flag is included in *flags*.)

  FTW_SL — The object is a symbolic link. (This condition shall only occur if the FTW_PHYS flag is included in *flags*.)

  FTW_SLN — The object is a symbolic link that does not name an existing file. (This condition shall only occur if the FTW_PHYS flag is not included in *flags*.)

  FTW_DNR — The object is a directory that cannot be read. The *fn* function shall not be called for any of its descendants.

  FTW_NS — The *stat*() function failed on the object because of lack of appropriate permission. The **stat** buffer passed to *fn* is undefined. Failure of *stat*() for any other reason is considered an error and *nftw*() shall return –1.

- The fourth argument is a pointer to an **FTW** structure. The value of **base** is the offset of the object's filename in the pathname passed as the first argument to *fn*. The value of **level** indicates depth relative to the root of the walk, where the root level is 0.

The results are unspecified if the application-supplied *fn* function does not preserve the current working directory.

The argument *depth* sets the maximum number of file descriptors that are shall be used by *nftw*( ) while traversing the file tree. At most one file descriptor shall be used for each directory level.

The *nftw*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *nftw*( ) function shall continue until the first of the following conditions occurs:

- An invocation of *fn* shall return a non-zero value, in which case *nftw*( ) shall return that value.
- The *nftw*( ) function detects an error other than [EACCES] (see FTW_DNR and FTW_NS above), in which case *nftw*( ) shall return –1 and set *errno* to indicate the error.
- The tree is exhausted, in which case *nftw*( ) shall return 0.

ERRORS

The *nftw*( ) function shall fail if:

[EACCES] Search permission is denied for any component of *path* or read permission is denied for *path*, or *fn* returns –1 and does not reset *errno*.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[ENAMETOOLONG]
The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

[ENOTDIR] A component of *path* is not a directory.

[EOVERFLOW] A field in the **stat** structure cannot be represented correctly in the current programming environment for one or more files found in the file hierarchy.

The *nftw*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

[ENAMETOOLONG]
Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

[ENFILE] Too many files are currently open in the system.

In addition, *errno* may be set if the function pointed by *fn* causes *errno* to be set.

EXAMPLES

The following example walks the **/tmp** directory and its subdirectories, calling the *nftw*() function for every directory entry, to a maximum of 5 levels deep.

```
#include <ftw.h>
...
int nftwfunc(const char *, const struct stat *, int, struct FTW *);

int nftwfunc(const char *filename, const struct stat *statptr,
    int fileflags, struct FTW *pfwt)
{
    return 0;
}
...
char *startpath = "/tmp";
int depth = 5;
int flags = FTW_CHDIR | FTW_DEPTH | FTW_MOUNT;
int ret;

ret = nftw(startpath, nftwfunc, depth, flags);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

lstat( ), *opendir*( ), *readdir*( ), *stat*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ftw.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

In the DESCRIPTION, the definition of the *depth* argument is clarified.

Issue 6

The Open Group Base Resolution bwg97-003 is applied.

The ERRORS section is updated as follows:

- The wording of the mandatory [ELOOP] error condition is updated.
- A second optional [ELOOP] error condition is added.
- The [EOVERFLOW] mandatory error condition is added.

Text is added to the DESCRIPTION to say that the *nftw*() function need not be reentrant and that the results are unspecified if the application-supplied *fn* function does not preserve the current working directory.

27106 **NAME**

27107 nice — change the nice value of a process

27108 **SYNOPSIS**

27109 XSI
```
#include <unistd.h>

int nice(int incr);
```
27110
27111

27112 **DESCRIPTION**

27113 The *nice*( ) function shall add the value of *incr* to the nice value of the calling process. A process'
27114 nice value is a non-negative number for which a more positive value shall result in less favorable
27115 scheduling.

27116 A maximum nice value of 2*{NZERO}–1 and a minimum nice value of 0 shall be imposed by the |
27117 system. Requests for values above or below these limits shall result in the nice value being set to |
27118 the corresponding limit. Only a process with appropriate privileges can lower the nice value. |

27119 PS | TPS Calling the *nice*( ) function has no effect on the priority of processes or threads with policy
27120 SCHED_FIFO or SCHED_RR. The effect on processes or threads with other scheduling policies
27121 is implementation-defined.

27122 The nice value set with *nice*( ) shall be applied to the process. If the process is multi-threaded, the
27123 nice value shall affect all system scope threads in the process.

27124 As –1 is a permissible return value in a successful situation, an application wishing to check for
27125 error situations should set *errno* to 0, then call *nice*( ), and if it returns –1, check to see whether |
27126 *errno* is non-zero.

27127 **RETURN VALUE**

27128 Upon successful completion, *nice*( ) shall return the new nice value –{NZERO}. Otherwise, –1
27129 shall be returned, the process' nice value shall not be changed, and *errno* shall be set to indicate
27130 the error.

27131 **ERRORS**

27132 The *nice*( ) function shall fail if:

27133 [EPERM] The *incr* argument is negative and the calling process does not have
27134 appropriate privileges.

27135 **EXAMPLES**

27136 **Changing the Nice Value**

27137 The following example adds the value of the *incr* argument, –20, to the nice value of the calling
27138 process.

```
27139 #include <unistd.h>
27140 ...
27141 int incr = -20;
27142 int ret;

27143 ret = nice(incr);
```

27144 **APPLICATION USAGE**

27145 None.

27146 **RATIONALE**

27147 None.

27148 **FUTURE DIRECTIONS**

27149 None.

27150 **SEE ALSO**

27151 *getpriority*( ), *setpriority*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**,
27152 **<unistd.h>**

27153 **CHANGE HISTORY**

27154 First released in Issue 1. Derived from Issue 1 of the SVID.

27155 **Issue 5**

27156 A statement is added to the description indicating the effects of this function on the different
27157 scheduling policies and multi-threaded processes.

27158 **NAME**

27159 nl_langinfo — language information

27160 **SYNOPSIS**

27161 XSI
```
#include <langinfo.h>

char *nl_langinfo(nl_item item);
```
27162

27163

27164 **DESCRIPTION**

27165 The *nl_langinfo*( ) function shall return a pointer to a string containing information relevant to
27166 the particular language or cultural area defined in the program's locale (see **<langinfo.h>**). The
27167 manifest constant names and values of *item* are defined in **<langinfo.h>**. For example:

27168
```
nl_langinfo(ABDAY_1)
```
27169 would return a pointer to the string `"Dom"` if the identified language was Portuguese, and
27170 `"Sun"` if the identified language was English.

27171 Calls to *setlocale*( ) with a category corresponding to the category of *item* (see **<langinfo.h>**), or to
27172 the category *LC_ALL*, may overwrite the array pointed to by the return value.

27173 The *nl_langinfo*( ) function need not be reentrant. A function that is not required to be reentrant is
27174 not required to be thread-safe.

27175 **RETURN VALUE**

27176 In a locale where *langinfo* data is not defined, *nl_langinfo*( ) shall return a pointer to the
27177 corresponding string in the POSIX locale. In all locales, *nl_langinfo*( ) shall return a pointer to an
27178 empty string if *item* contains an invalid setting.

27179 This pointer may point to static data that may be overwritten on the next call.

27180 **ERRORS**

27181 No errors are defined.

27182 **EXAMPLES**

27183 **Getting Date and Time Formatting Information**

27184 The following example returns a pointer to a string containing date and time formatting
27185 information, as defined in the *LC_TIME* category of the current locale.

27186
```
#include <time.h>
#include <langinfo.h>
...
strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);
...
```
27187

27188

27189

27190

27191 **APPLICATION USAGE**

27192 The array pointed to by the return value should not be modified by the program, but may be
27193 modified by further calls to *nl_langinfo*( ).

27194 **RATIONALE**

27195 None.

27196 **FUTURE DIRECTIONS**

27197 None.

27198 **SEE ALSO**

27199 *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<langinfo.h>**, **<nl_types.h>**, the
27200 Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

27201 **CHANGE HISTORY**

27202 First released in Issue 2.

27203 **Issue 5**

27204 The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section.

27205 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

NAME

nrand48 — generate uniformly distributed pseudo-random non-negative long integers

SYNOPSIS

XSI

```
#include <stdlib.h>

long nrand48(unsigned short xsubi[3]);
```

DESCRIPTION

Refer to *drand48*( ).

27214 **NAME**

27215 ntohl — convert values between host and network byte order

27216 **SYNOPSIS**

27217 `#include <arpa/inet.h>`

27218 `uint32_t ntohl(uint32_t` *`netlong`*`);`

27219 **DESCRIPTION**

27220 Refer to *htonl*( ).

27221 **NAME**

27222 ntohs — convert values between host and network byte order

27223 **SYNOPSIS**

27224 `#include <arpa/inet.h>`

27225 `uint16_t ntohs(uint16_t` *`netshort`*`);`

27226 **DESCRIPTION**

27227 Refer to *htonl*( ).

27228 **NAME**

27229 open — open a file

27230 **SYNOPSIS**

27231 OH
```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int oflag, ... );
```

27234 **DESCRIPTION**

27235 The *open*( ) function shall establish the connection between a file and a file descriptor. It shall
27236 create an open file description that refers to a file and a file descriptor that refers to that open file
27237 description. The file descriptor is used by other I/O functions to refer to that file. The *path* |
27238 argument points to a pathname naming the file. |

27239 The *open*( ) function shall return a file descriptor for the named file that is the lowest file
27240 descriptor not currently open for that process. The open file description is new, and therefore the |
27241 file descriptor shall not share it with any other process in the system. The FD_CLOEXEC file |
27242 descriptor flag associated with the new file descriptor shall be cleared.

27243 The file offset used to mark the current position within the file shall be set to the beginning of the
27244 file.

27245 The file status flags and file access modes of the open file description shall be set according to
27246 the value of *oflag*.

27247 Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list,
27248 defined in **<fcntl.h>**. Applications shall specify exactly one of the first three values (file access
27249 modes) below in the value of *oflag*:

27250 O_RDONLY Open for reading only.

27251 O_WRONLY Open for writing only.

27252 O_RDWR Open for reading and writing. The result is undefined if this flag is applied to
27253 a FIFO.

27254 Any combination of the following may be used:

27255 O_APPEND If set, the file offset shall be set to the end of the file prior to each write.

27256 O_CREAT If the file exists, this flag has no effect except as noted under O_EXCL below. |
27257 Otherwise, the file shall be created; the user ID of the file shall be set to the |
27258 effective user ID of the process; the group ID of the file shall be set to the |
27259 group ID of the file's parent directory or to the effective group ID of the |
27260 process; and the access permission bits (see **<sys/stat.h>**) of the file mode shall |
27261 be set to the value of the third argument taken as type **mode_t** modified as
27262 follows: a bitwise AND is performed on the file-mode bits and the
27263 corresponding bits in the complement of the process' file mode creation mask.
27264 Thus, all bits in the file mode whose corresponding bit in the file mode |
27265 creation mask is set are cleared. When bits other than the file permission bits |
27266 are set, the effect is unspecified. The third argument does not affect whether |
27267 the file is open for reading, writing, or for both. Implementations shall provide |
27268 a way to initialize the file's group ID to the group ID of the parent directory. |
27269 Implementations may, but need not, provide an implementation-defined way |
27270 to initialize the file's group ID to the effective group ID of the calling process. |

27271 SIO O_DSYNC Write I/O operations on the file descriptor shall complete as defined by |
27272 synchronized I/O data integrity completion. |

O_EXCL — If O_CREAT and O_EXCL are set, *open*( ) shall fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist shall be atomic with respect to other threads executing *open*( ) naming the same filename in the same directory with O_EXCL and O_CREAT set. If O_EXCL and O_CREAT are set, and *path* names a symbolic link, *open*( ) shall fail and set *errno* to [EEXIST], regardless of the contents of the symbolic link. If O_EXCL is set and O_CREAT is not set, the result is undefined.

O_NOCTTY — If set and *path* identifies a terminal device, *open*( ) shall not cause the terminal device to become the controlling terminal for the process.

O_NONBLOCK — When opening a FIFO with O_RDONLY or O_WRONLY set:

- If O_NONBLOCK is set, an *open*( ) for reading-only shall return without delay. An *open*( ) for writing-only shall return an error if no process currently has the file open for reading.
- If O_NONBLOCK is clear, an *open*( ) for reading-only shall block the calling thread until a thread opens the file for writing. An *open*( ) for writing-only shall block the calling thread until a thread opens the file for reading.

When opening a block special or character special file that supports non-blocking opens:

- If O_NONBLOCK is set, the *open*( ) function shall return without blocking for the device to be ready or available. Subsequent behavior of the device is device-specific.
- If O_NONBLOCK is clear, the *open*( ) function shall block the calling thread until the device is ready or available before returning.

Otherwise, the behavior of O_NONBLOCK is unspecified.

SIO O_RSYNC — Read I/O operations on the file descriptor shall complete at the same level of integrity as specified by the O_DSYNC and O_SYNC flags. If both O_DSYNC and O_RSYNC are set in *oflag*, all I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion. If both O_SYNC and O_RSYNC are set in flags, all I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

SIO O_SYNC — Write I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

O_TRUNC — If the file exists and is a regular file, and the file is successfully opened O_RDWR or O_WRONLY, its length shall be truncated to 0, and the mode and owner shall be unchanged. It shall have no effect on FIFO special files or terminal device files. Its effect on other file types is implementation-defined. The result of using O_TRUNC with O_RDONLY is undefined.

If O_CREAT is set and the file did not previously exist, upon successful completion, *open*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file and the *st_ctime* and *st_mtime* fields of the parent directory.

If O_TRUNC is set and the file did previously exist, upon successful completion, *open*( ) shall mark for update the *st_ctime* and *st_mtime* fields of the file.

27317 SIO If both the O_SYNC and O_DSYNC flags are set, the effect is as if only the O_SYNC flag was set.
27318

27319 XSR If *path* refers to a STREAMS file, *oflag* may be constructed from O_NONBLOCK OR'ed with
27320 either O_RDONLY, O_WRONLY, or O_RDWR. Other flag values are not applicable to
27321 STREAMS devices and shall have no effect on them. The value O_NONBLOCK affects the
27322 operation of STREAMS drivers and certain functions applied to file descriptors associated with
27323 STREAMS files. For STREAMS drivers, the implementation of O_NONBLOCK is device-specific.
27324

27325 XSI If *path* names the master side of a pseudo-terminal device, then it is unspecified whether *open*( )
27326 locks the slave side so that it cannot be opened. Conforming applications shall call *unlockpt*( )
27327 before opening the slave side.

27328 The largest value that can be represented correctly in an object of type **off_t** shall be established
27329 as the offset maximum in the open file description.

27330 **RETURN VALUE**

27331 Upon successful completion, the function shall open the file and return a non-negative integer
27332 representing the lowest numbered unused file descriptor. Otherwise, –1 shall be returned and
27333 *errno* set to indicate the error. No files shall be created or modified if the function returns –1.

27334 **ERRORS**

27335 The *open*( ) function shall fail if:

27336 [EACCES] Search permission is denied on a component of the path prefix, or the file
27337 exists and the permissions specified by *oflag* are denied, or the file does not
27338 exist and write permission is denied for the parent directory of the file to be
27339 created, or O_TRUNC is specified and write permission is denied.

27340 [EEXIST] O_CREAT and O_EXCL are set, and the named file exists.

27341 [EINTR] A signal was caught during *open*( ).

27342 SIO [EINVAL] The implementation does not support synchronized I/O for this file.

27343 XSR [EIO] The *path* argument names a STREAMS file and a hangup or error occurred
27344 during the *open*( ).

27345 [EISDIR] The named file is a directory and *oflag* includes O_WRONLY or O_RDWR.

27346 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
27347 argument.

27348 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

27349 [ENAMETOOLONG]
27350 The length of the *path* argument exceeds {PATH_MAX} or a pathname
27351 component is longer than {NAME_MAX}.

27352 [ENFILE] The maximum allowable number of files is currently open in the system.

27353 [ENOENT] O_CREAT is not set and the named file does not exist; or O_CREAT is set and
27354 either the path prefix does not exist or the *path* argument points to an empty
27355 string.

27356 XSR [ENOSR] The *path* argument names a STREAMS-based file and the system is unable to
27357 allocate a STREAM.

27358 [ENOSPC] The directory or file system that would contain the new file cannot be
27359 expanded, the file does not exist, and O_CREAT is specified.

27360 [ENOTDIR] A component of the path prefix is not a directory.

27361 [ENXIO] O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set, and no
27362 process has the file open for reading.

27363 [ENXIO] The named file is a character special or block special file, and the device
27364 associated with this special file does not exist.

27365 [EOVERFLOW] The named file is a regular file and the size of the file cannot be represented
27366 correctly in an object of type **off_t**.

27367 [EROFS] The named file resides on a read-only file system and either O_WRONLY,
27368 O_RDWR, O_CREAT (if file does not exist), or O_TRUNC is set in the *oflag*
27369 argument.

27370 The *open*( ) function may fail if:

27371 XSI [EAGAIN] The *path* argument names the slave side of a pseudo-terminal device that is
27372 locked.

27373 [EINVAL] The value of the *oflag* argument is not valid.

27374 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
27375 resolution of the *path* argument.

27376 [ENAMETOOLONG]
27377 As a result of encountering a symbolic link in resolution of the *path* argument, |
27378 the length of the substituted pathname string exceeded {PATH_MAX}. |

27379 XSR [ENOMEM] The *path* argument names a STREAMS file and the system is unable to allocate
27380 resources.

27381 [ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *oflag* is
27382 O_WRONLY or O_RDWR.

27383 **EXAMPLES**

27384 **Opening a File for Writing by the Owner**

27385 The following example opens the file **/tmp/file**, either by creating it (if it does not already exist),
27386 or by truncating its length to 0 (if it does exist). In the former case, if the call creates a new file,
27387 the access permission bits in the file mode of the file are set to permit reading and writing by the
27388 owner, and to permit reading only by group members and others.

27389 If the call to *open*( ) is successful, the file is opened for writing.

```
27390 #include <fcntl.h>
27391 ...
27392 int fd;
27393 mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
27394 char *filename = "/tmp/file";
27395 ...
27396 fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, mode);
27397 ...
```

Opening a File Using an Existence Check

The following example uses the *open*() function to try to create the **LOCKFILE** file and open it for writing. Since the *open*() function specifies the O_EXCL flag, the call fails if the file already exists. In that case, the program assumes that someone else is updating the password file and exits.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define LOCKFILE "/etc/ptmp"
...
int pfd; /* Integer for file descriptor returned by open() call. */
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}
...
```

Opening a File for Writing

The following example opens a file for writing, creating the file if it does not already exist. If the file does exist, the system truncates the file to zero bytes.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define LOCKFILE "/etc/ptmp"
...
int pfd;
char filename[PATH_MAX+1];
...
if ((pfd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    perror("Cannot open output file\n"); exit(1);
}
...
```

APPLICATION USAGE

None.

RATIONALE

Except as specified in this volume of IEEE Std 1003.1-200x, the flags allowed in *oflag* are not mutually-exclusive and any number of them may be used simultaneously.

Some implementations permit opening FIFOs with O_RDWR. Since FIFOs could be implemented in other ways, and since two file descriptors can be used to the same effect, this possibility is left as undefined.

See *getgroups*() about the group of a newly created file.

The use of *open*( ) to create a regular file is preferable to the use of *creat*( ), because the latter is redundant and included only for historical reasons.

The use of the O_TRUNC flag on FIFOs and directories (pipes cannot be *open*( )-ed) must be permissible without unexpected side effects (for example, *creat*( ) on a FIFO must not remove data). Since terminal special files might have type-ahead data stored in the buffer, O_TRUNC should not affect their content, particularly if a program that normally opens a regular file should open the current controlling terminal instead. Other file types, particularly implementation-defined ones, are left implementation-defined.

IEEE Std 1003.1-200x permits [EACCES] to be returned for conditions other than those explicitly listed.

The O_NOCTTY flag was added to allow applications to avoid unintentionally acquiring a controlling terminal as a side effect of opening a terminal file. This volume of IEEE Std 1003.1-200x does not specify how a controlling terminal is acquired, but it allows an implementation to provide this on *open*( ) if the O_NOCTTY flag is not set and other conditions specified in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface are met. The O_NOCTTY flag is an effective no-op if the file being opened is not a terminal device.

In historical implementations the value of O_RDONLY is zero. Because of that, it is not possible to detect the presence of O_RDONLY and another option. Future implementations should encode O_RDONLY and O_WRONLY as bit flags so that:

```
O_RDONLY | O_WRONLY == O_RDWR
```

In general, the *open*( ) function follows the symbolic link if *path* names a symbolic link. However, the *open*( ) function, when called with O_CREAT and O_EXCL, is required to fail with [EEXIST] if *path* names an existing symbolic link, even if the symbolic link refers to a nonexistent file. This behavior is required so that privileged applications can create a new file in a known location without the possibility that a symbolic link might cause the file to be created in a different location.

For example, a privileged application that must create a file with a predictable name in a user-writable directory, such as the user's home directory, could be compromised if the user creates a symbolic link with that name that refers to a nonexistent file in a system directory. If the user can influence the contents of a file, the user could compromise the system by creating a new system configuration or spool file that would then be interpreted by the system. The test for a symbolic link which refers to a nonexisting file must be atomic with the creation of a new file.

The POSIX.1-1990 standard required that the group ID of a newly created file be set to the group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required that implementations provide a way to have the group ID be set to the group ID of the containing directory, but did not prohibit implementations also supporting a way to set the group ID to the effective group ID of the creating process. Conforming applications should not assume which group ID will be used. If it matters, an application can use *chown*( ) to set the group ID after the file is created, or determine under what conditions the implementation will set the desired group ID.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *close*( ), *creat*( ), *dup*( ), *fcntl*( ), *lseek*( ), *read*( ), *umask*( ), *unlockpt*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the DESCRIPTION, O_CREAT is amended to state that the group ID of the file is set to the group ID of the file's parent directory or to the effective group ID of the process. This is a FIPS requirement.
- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.
- In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support large files.
- The [ENXIO] mandatory error condition is added.
- The [EINVAL], [ENAMETOOLONG], and [ETXTBSY] optional error conditions are added.

The DESCRIPTION and ERRORS sections are updated so that items related to the optional XSI STREAMS Option Group are marked.

The following changes were made to align with the IEEE P1003.1a draft standard:

- An explanation is added of the effect of the O_CREAT and O_EXCL flags when the path refers to a symbolic link.
- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

The DESCRIPTION of O_EXCL is updated in response to IEEE PASC Interpretation 1003.1c #48.

27519 **NAME**

27520 opendir — open a directory

27521 **SYNOPSIS**

27522 `#include <dirent.h>`

27523 `DIR *opendir(const char *dirname);`

27524 **DESCRIPTION**

27525 The *opendir*( ) function shall open a directory stream corresponding to the directory named by
27526 the *dirname* argument. The directory stream is positioned at the first entry. If the type **DIR** is
27527 implemented using a file descriptor, applications shall only be able to open up to a total of
27528 {OPEN_MAX} files and directories.

27529 **RETURN VALUE**

27530 Upon successful completion, *opendir*( ) shall return a pointer to an object of type **DIR**.
27531 Otherwise, a null pointer shall be returned and *errno* set to indicate the error.

27532 **ERRORS**

27533 The *opendir*( ) function shall fail if:

27534 [EACCES] Search permission is denied for the component of the path prefix of *dirname* or
27535 read permission is denied for *dirname*.

27536 [ELOOP] A loop exists in symbolic links encountered during resolution of the *dirname*
27537 argument.

27538 [ENAMETOOLONG]
27539 The length of the *dirname* argument exceeds {PATH_MAX} or a pathname |
27540 component is longer than {NAME_MAX}. |

27541 [ENOENT] A component of *dirname* does not name an existing directory or *dirname* is an
27542 empty string.

27543 [ENOTDIR] A component of *dirname* is not a directory.

27544 The *opendir*( ) function may fail if:

27545 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
27546 resolution of the *dirname* argument.

27547 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

27548 [ENAMETOOLONG]
27549 As a result of encountering a symbolic link in resolution of the *dirname* |
27550 argument, the length of the substituted pathname string exceeded |
27551 {PATH_MAX}.

27552 [ENFILE] Too many files are currently open in the system.

EXAMPLES

Open a Directory Stream

The following program fragment demonstrates how the *opendir*( ) function is used.

```
#include <sys/types.h>
#include <dirent.h>
#include <libgen.h>
...
    DIR *dir;
    struct dirent *dp;
...
    if ((dir = opendir (".")) == NULL) {
        perror ("Cannot open .");
        exit (1);
    }

    while ((dp = readdir (dir)) != NULL) {
...
```

APPLICATION USAGE

The *opendir*( ) function should be used in conjunction with *readdir*( ), *closedir*( ), and *rewinddir*( ) to examine the contents of the directory (see the EXAMPLES section in *readdir*( )). This method is recommended for portability.

RATIONALE

Based on historical implementations, the rules about file descriptors apply to directory streams as well. However, this volume of IEEE Std 1003.1-200x does not mandate that the directory stream be implemented using file descriptors. The description of *closedir*( ) clarifies that if a file descriptor is used for the directory stream, it is mandatory that *closedir*( ) deallocate the file descriptor. When a file descriptor is used to implement the directory stream, it behaves as if the FD_CLOEXEC had been set for the file descriptor.

The directory entries for dot and dot-dot are optional. This volume of IEEE Std 1003.1-200x does not provide a way to test *a priori* for their existence because an application that is portable must be written to look for (and usually ignore) those entries. Writing code that presumes that they are the first two entries does not always work, as many implementations permit them to be other than the first two entries, with a ‘‘normal’’ entry preceding them. There is negligible value in providing a way to determine what the implementation does because the code to deal with dot and dot-dot must be written in any case and because such a flag would add to the list of those flags (which has proven in itself to be objectionable) and might be abused.

Since the structure and buffer allocation, if any, for directory operations are defined by the implementation, this volume of IEEE Std 1003.1-200x imposes no portability requirements for erroneous program constructs, erroneous data, or the use of unspecified values such as the use or referencing of a *dirp* value or a **dirent** structure value after a directory stream has been closed or after a *fork*( ) or one of the *exec* function calls.

FUTURE DIRECTIONS

None.

SEE ALSO

closedir( ), *lstat*( ), *readdir*( ), *rewinddir*( ), *symlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**, **<limits.h>**, **<sys/types.h>**

27598 **CHANGE HISTORY**

27599 First released in Issue 2.

27600 **Issue 6**

27601 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

27602 The following new requirements on POSIX implementations derive from alignment with the | 27603 Single UNIX Specification:

27604 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was 27605 required for conforming implementations of previous POSIX specifications, it was not 27606 required for UNIX applications.

27607 • The [ELOOP] mandatory error condition is added.

27608 • A second [ENAMETOOLONG] is added as an optional error condition.

27609 The following changes were made to align with the IEEE P1003.1a draft standard:

27610 • The [ELOOP] optional error condition is added.

27611 **NAME**

27612 openlog — open a connection to the logging facility

27613 **SYNOPSIS**

27614 XSI
```
#include <syslog.h>

void openlog(const char *ident, int logopt, int facility);
```
27615

27616

27617 **DESCRIPTION**

27618 Refer to *closelog*( ).

27619 **NAME**

27620 optarg, opterr, optind, optopt — options parsing variables

27621 **SYNOPSIS**

27622
```
#include <unistd.h>
```

27623
```
extern char *optarg;
```
27624
```
extern int opterr, optind, optopt;
```

27625 **DESCRIPTION**

27626 Refer to *getopt*( ).

27627 **NAME**

27628 pathconf — get configurable pathname variables

27629 **SYNOPSIS**

27630 `#include <unistd.h>`

27631 `long pathconf(const char *path, int name);`

27632 **DESCRIPTION**

27633 Refer to *fpathconf*( ).

27634 **NAME**

27635 pause — suspend the thread until a signal is received

27636 **SYNOPSIS**

27637
```
#include <unistd.h>
```

27638
```
int pause(void);
```

27639 **DESCRIPTION**

27640 The *pause*( ) function shall suspend the calling thread until delivery of a signal whose action is
27641 either to execute a signal-catching function or to terminate the process.

27642 If the action is to terminate the process, *pause*( ) shall not return.

27643 If the action is to execute a signal-catching function, *pause*( ) shall return after the signal-catching
27644 function returns.

27645 **RETURN VALUE**

27646 Since *pause*( ) suspends thread execution indefinitely unless interrupted by a signal, there is no
27647 successful completion return value. A value of –1 shall be returned and *errno* set to indicate the
27648 error.

27649 **ERRORS**

27650 The *pause*( ) function shall fail if:

27651 [EINTR] A signal is caught by the calling process and control is returned from the
27652 signal-catching function.

27653 **EXAMPLES**

27654 None.

27655 **APPLICATION USAGE**

27656 Many common uses of *pause*( ) have timing windows. The scenario involves checking a
27657 condition related to a signal and, if the signal has not occurred, calling *pause*( ). When the signal
27658 occurs between the check and the call to *pause*( ), the process often blocks indefinitely. The
27659 *sigprocmask*( ) and *sigsuspend*( ) functions can be used to avoid this type of problem.

27660 **RATIONALE**

27661 None.

27662 **FUTURE DIRECTIONS**

27663 None.

27664 **SEE ALSO**

27665 *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

27666 **CHANGE HISTORY**

27667 First released in Issue 1. Derived from Issue 1 of the SVID.

27668 **Issue 5**

27669 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

27670 **Issue 6**

27671 The APPLICATION USAGE section is added.

NAME

pclose — close a pipe stream to or from a process

SYNOPSIS

CX
```
#include <stdio.h>

int pclose(FILE *stream);
```

DESCRIPTION

The *pclose*( ) function shall close a stream that was opened by *popen*( ), wait for the command to terminate, and return the termination status of the process that was running the command language interpreter. However, if a call caused the termination status to be unavailable to *pclose*( ), then *pclose*( ) shall return –1 with *errno* set to [ECHILD] to report this situation. This can happen if the application calls one of the following functions:

- *wait*( )
- *waitpid*( ) with a *pid* argument less than or equal to 0 or equal to the process ID of the command line interpreter
- Any other function not defined in this volume of IEEE Std 1003.1-200x that could do one of the above

In any case, *pclose*( ) shall not return before the child process created by *popen*( ) has terminated.

If the command language interpreter cannot be executed, the child termination status returned by *pclose*( ) shall be as if the command language interpreter terminated using *exit*(127) or *_exit*(127).

The *pclose*( ) function shall not affect the termination status of any child of the calling process other than the one created by *popen*( ) for the associated stream.

If the argument *stream* to *pclose*( ) is not a pointer to a stream created by *popen*( ), the result of *pclose*( ) is undefined.

RETURN VALUE

Upon successful return, *pclose*( ) shall return the termination status of the command language interpreter. Otherwise, *pclose*( ) shall return –1 and set *errno* to indicate the error.

ERRORS

The *pclose*( ) function shall fail if:

[ECHILD] The status of the child process could not be obtained, as described above.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

There is a requirement that *pclose*( ) not return before the child process terminates. This is intended to disallow implementations that return [EINTR] if a signal is received while waiting. If *pclose*( ) returned before the child terminated, there would be no way for the application to discover which child used to be associated with the stream, and it could not do the cleanup itself.

If the stream pointed to by *stream* was not created by *popen*( ), historical implementations of *pclose*( ) return –1 without setting *errno*. To avoid requiring *pclose*( ) to set *errno* in this case, IEEE Std 1003.1-200x makes the behavior unspecified. An application should not use *pclose*( ) to

close any stream that was not created by *popen*( ).

Some historical implementations of *pclose*( ) either block or ignore the signals SIGINT, SIGQUIT, and SIGHUP while waiting for the child process to terminate. Since this behavior is not described for the *pclose*( ) function in IEEE Std 1003.1-200x, such implementations are not conforming. Also, some historical implementations return [EINTR] if a signal is received, even though the child process has not terminated. Such implementations are also considered non-conforming.

Consider, for example, an application that uses:

```
popen("command", "r")
```

to start *command*, which is part of the same application. The parent writes a prompt to its standard output (presumably the terminal) and then reads from the stream. The child reads the response from the user, does some transformation on the response (pathname expansion, perhaps) and writes the result to its standard output. The parent process reads the result from the pipe, does something with it, and prints another prompt. The cycle repeats. Assuming that both processes do appropriate buffer flushing, this would be expected to work.

To conform to IEEE Std 1003.1-200x, *pclose*( ) must use *waitpid*( ), or some similar function, instead of *wait*( ).

The code sample below illustrates how the *pclose*( ) function might be implemented on a system conforming to IEEE Std 1003.1-200x.

```
int pclose(FILE *stream)
{
    int stat;
    pid_t pid;

    pid = <pid for process created for stream by popen()>
    (void) fclose(stream);
    while (waitpid(pid, &stat, 0) == -1) {
        if (errno != EINTR){
            stat = -1;
            break;
        }
    }
    return(stat);
}
```

FUTURE DIRECTIONS

None.

SEE ALSO

fork( ), *popen*( ), *waitpid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

perror — write error messages to standard error

SYNOPSIS

```
#include <stdio.h>

void perror(const char *s);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *perror*( ) function shall map the error number accessed through the symbol *errno* to a language-dependent error message, which shall be written to the standard error stream as follows:

- First (if *s* is not a null pointer and the character pointed to by *s* is not the null byte), the string pointed to by *s* followed by a colon and a <space>.
- Then an error message string followed by a <newline>.

The contents of the error message strings shall be the same as those returned by *strerror*( ) with argument *errno*.

CX The *perror*( ) function shall mark the file associated with the standard error stream as having been written (*st_ctime*, *st_mtime* marked for update) at some time between its successful completion and *exit*( ), *abort*( ), or the completion of *fflush*( ) or *fclose*( ) on *stderr*.

The *perror*( ) function shall not change the orientation of the standard error stream.

RETURN VALUE

The *perror*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

Printing an Error Message for a Function

The following example replaces *bufptr* with a buffer that is the necessary size. If an error occurs, the *perror*( ) function prints a message and the program exits.

```
#include <stdio.h>
#include <stdlib.h>
...
char *bufptr;
size_t szbuf;
...
if ((bufptr = malloc(szbuf)) == NULL) {
    perror("malloc"); exit(2);
}
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strerror( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A paragraph is added to the DESCRIPTION indicating that *perror*( ) does not change the orientation of the standard error stream.

Issue 6

Extensions beyond the ISO C standard are now marked.

27809 **NAME**

27810 pipe — create an interprocess channel

27811 **SYNOPSIS**

27812 `#include <unistd.h>`

27813 `int pipe(int fildes[2]);`

27814 **DESCRIPTION**

27815 The *pipe*( ) function shall create a pipe and place two file descriptors, one each into the
27816 arguments *fildes*[0] and *fildes*[1], that refer to the open file descriptions for the read and write
27817 ends of the pipe. Their integer values shall be the two lowest available at the time of the *pipe*( )
27818 call. The O_NONBLOCK and FD_CLOEXEC flags shall be clear on both file descriptors. (The
27819 *fcntl*( ) function can be used to set both these flags.)

27820 Data can be written to the file descriptor *fildes*[1] and read from the file descriptor *fildes*[0]. A
27821 read on the file descriptor *fildes*[0] shall access data written to the file descriptor *fildes*[1] on a
27822 first-in-first-out basis. It is unspecified whether *fildes*[0] is also open for writing and whether
27823 *fildes*[1] is also open for reading.

27824 A process has the pipe open for reading (correspondingly writing) if it has a file descriptor open
27825 that refers to the read end, *fildes*[0] (write end, *fildes*[1]).

27826 Upon successful completion, *pipe*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
27827 fields of the pipe.

27828 **RETURN VALUE**

27829 Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to
27830 indicate the error.

27831 **ERRORS**

27832 The *pipe*( ) function shall fail if:

27833 [EMFILE] More than {OPEN_MAX} minus two file descriptors are already in use by this
27834 process.

27835 [ENFILE] The number of simultaneously open files in the system would exceed a
27836 system-imposed limit.

27837 **EXAMPLES**

27838 None.

27839 **APPLICATION USAGE**

27840 None.

27841 **RATIONALE**

27842 The wording carefully avoids using the verb ‘‘to open’’ in order to avoid any implication of use
27843 of *open*( ); see also *write*( ).

27844 **FUTURE DIRECTIONS**

27845 None.

27846 **SEE ALSO**

27847 *fcntl*( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**,
27848 **<unistd.h>**

27849 **CHANGE HISTORY**

27850 First released in Issue 1. Derived from Issue 1 of the SVID.

27851 **Issue 6**

27852 The following new requirements on POSIX implementations derive from alignment with the
27853 Single UNIX Specification:

27854 • The DESCRIPTION is updated to indicate that certain dispositions of *fildes*[0] and *fildes*[1]
27855 are unspecified.

NAME

poll — input/output multiplexing

SYNOPSIS

XSI
```
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

DESCRIPTION

The *poll*( ) function provides applications with a mechanism for multiplexing input/output over a set of file descriptors. For each member of the array pointed to by *fds*, *poll*( ) shall examine the given file descriptor for the event(s) specified in *events*. The number of **pollfd** structures in the *fds* array is specified by *nfds*. The *poll*( ) function shall identify those file descriptors on which an application can read or write data, or on which certain events have occurred.

The *fds* argument specifies the file descriptors to be examined and the events of interest for each file descriptor. It is a pointer to an array with one member for each open file descriptor of interest. The array's members are **pollfd** structures within which *fd* specifies an open file descriptor and *events* and *revents* are bitmasks constructed by OR'ing a combination of the following event flags:

POLLIN
: Data other than high-priority data may be read without blocking.

XSR
: For STREAMS, this flag is set in *revents* even if the message is of zero length. This flag shall be equivalent to POLLRDNORM | POLLRDBAND.

POLLRDNORM
: Normal data may be read without blocking.

XSR
: For STREAMS, data on priority band 0 may be read without blocking. This flag is set in *revents* even if the message is of zero length.

POLLRDBAND
: Priority data may be read without blocking.

XSR
: For STREAMS, data on priority bands greater than 0 may be read without blocking. This flag is set in *revents* even if the message is of zero length.

POLLPRI
: High-priority data may be read without blocking.

XSR
: For STREAMS, this flag is set in *revents* even if the message is of zero length.

POLLOUT
: Normal data may be written without blocking.

XSR
: For STREAMS, data on priority band 0 may be written without blocking.

POLLWRNORM
: Equivalent to POLLOUT.

POLLWRBAND
: Priority data may be written.

XSR
: For STREAMS, data on priority bands greater than 0 may be written without blocking. If any priority band has been written to on this STREAM, this event only examines bands that have been written to at least once.

POLLERR
: An error has occurred on the device or stream. This flag is only valid in the *revents* bitmask; it shall be ignored in the *events* member.

POLLHUP
: The device has been disconnected. This event and POLLOUT are mutually-exclusive; a stream can never be writable if a hangup has occurred. However, this event and POLLIN, POLLRDNORM, POLLRDBAND, or POLLPRI are not mutually-exclusive. This flag is only valid in the *revents* bitmask; it shall be ignored in the *events* member.

POLLNVAL The specified *fd* value is invalid. This flag is only valid in the *revents* member; it shall ignored in the *events* member.

The significance and semantics of normal, priority, and high-priority data are file and device-specific.

If the value of *fd* is less than 0, *events* shall be ignored, and *revents* shall be set to 0 in that entry on return from *poll*( ).

In each **pollfd** structure, *poll*( ) shall clear the *revents* member, except that where the application requested a report on a condition by setting one of the bits of *events* listed above, *poll*( ) shall set the corresponding bit in *revents* if the requested condition is true. In addition, *poll*( ) shall set the POLLHUP, POLLERR, and POLLNVAL flag in *revents* if the condition is true, even if the application did not set the corresponding bit in *events*.

If none of the defined events have occurred on any selected file descriptor, *poll*( ) shall wait at least *timeout* milliseconds for an event to occur on any of the selected file descriptors. If the value of *timeout* is 0, *poll*( ) shall return immediately. If the value of *timeout* is –1, *poll*( ) shall block until a requested event occurs or until the call is interrupted.

Implementations may place limitations on the granularity of timeout intervals. If the requested timeout interval requires a finer granularity than the implementation supports, the actual timeout interval shall be rounded up to the next supported value.

The *poll*( ) function shall not be affected by the O_NONBLOCK flag.

XSR The *poll*( ) function shall support regular files, terminal and pseudo-terminal devices, STREAMS-based files, FIFOs, pipes, and sockets. The behavior of *poll*( ) on elements of *fds* that refer to other types of file is unspecified.

Regular files shall always poll TRUE for reading and writing.

A file descriptor for a socket that is listening for connections shall indicate that it is ready for reading, once connections are available. A file descriptor for a socket that is connecting asynchronously shall indicate that it is ready for writing, once a connection has been established.

RETURN VALUE

Upon successful completion, *poll*( ) shall return a non-negative value. A positive value indicates the total number of file descriptors that have been selected (that is, file descriptors for which the *revents* member is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, *poll*( ) shall return –1 and set *errno* to indicate the error.

ERRORS

The *poll*( ) function shall fail if:

[EAGAIN] The allocation of internal data structures failed but a subsequent request may succeed.

[EINTR] A signal was caught during *poll*( ).

XSR [EINVAL] The *nfds* argument is greater than {OPEN_MAX}, or one of the *fd* members refers to a STREAM or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.

EXAMPLES

Checking for Events on a Stream

The following example opens a pair of STREAMS devices and then waits for either one to become writable. This example proceeds as follows:

1. Sets the *timeout* parameter to 500 milliseconds.
2. Opens the STREAMS devices **/dev/dev0** and **/dev/dev1**, and then polls them, specifying POLLOUT and POLLWRBAND as the events of interest.

   The STREAMS device names **/dev/dev0** and **/dev/dev1** are only examples of how STREAMS devices can be named; STREAMS naming conventions may vary among systems conforming to the IEEE Std 1003.1-200x.
3. Uses the *ret* variable to determine whether an event has occurred on either of the two STREAMS. The *poll*( ) function is given 500 milliseconds to wait for an event to occur (if it has not occurred prior to the *poll*( ) call).
4. Checks the returned value of *ret*. If a positive value is returned, one of the following can be done:
   a. Priority data can be written to the open STREAM on priority bands greater than 0, because the POLLWRBAND event occurred on the open STREAM (*fds*[0] or *fds*[1]).
   b. Data can be written to the open STREAM on priority-band 0, because the POLLOUT event occurred on the open STREAM (*fds*[0] or *fds*[1]).
5. If the returned value is not a positive value, permission to write data to the open STREAM (on any priority band) is denied.
6. If the POLLHUP event occurs on the open STREAM (*fds*[0] or *fds*[1]), the device on the open STREAM has disconnected.

```
#include <stropts.h>
#include <poll.h>
...
struct pollfd fds[2];
int timeout_msecs = 500;
int ret;
    int i;

/* Open STREAMS device. */
fds[0].fd = open("/dev/dev0", ...);
fds[1].fd = open("/dev/dev1", ...);
    fds[0].events = POLLOUT | POLLWRBAND;
    fds[1].events = POLLOUT | POLLWRBAND;

ret = poll(fds, 2, timeout_msecs);

if (ret > 0) {
    /* An event on one of the fds has occurred. */
    for (i=0; i<2; i++) {
        if (fds[i].revents & POLLWRBAND) {
        /* Priority data may be written on device number i. */
...
        }
        if (fds[i].revents & POLLOUT) {
```

```
                /* Data may be written on device number i. */
        ...
                }
                if (fds[i].revents & POLLHUP) {
                /* A hangup has occurred on device number i. */
        ...
                }
            }
        }
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getmsg( ), *putmsg*( ), *read*( ), *select*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<poll.h>**, **<stropts.h>**, Section 2.6 (on page 488)

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The description of POLLWRBAND is updated.

Issue 6

Text referring to sockets is added to the DESCRIPTION.

Text relating to the XSI STREAMS Option Group is marked. |

The Open Group Corrigendum Unnn/nn is applied, updating the DESCRIPTION of POLLWRBAND. | |

NAME

popen — initiate pipe streams to or from a process

SYNOPSIS

CX
```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
```

DESCRIPTION

The *popen*( ) function shall execute the command specified by the string *command*. It shall create a pipe between the calling program and the executed command, and shall return a pointer to a stream that can be used to either read from or write to the pipe.

The environment of the executed command shall be as if a child process were created within the *popen*( ) call using the *fork*( ) function, and the child invoked the *sh* utility using the call:

```
execl(shell path, "sh", "-c", command, (char *)0);
```

where *shell path* is an unspecified pathname for the *sh* utility.

The *popen*( ) function shall ensure that any streams from previous *popen*( ) calls that remain open in the parent process are closed in the new child process.

The *mode* argument to *popen*( ) is a string that specifies I/O mode:

1. If *mode* is *r*, when the child process is started, its file descriptor STDOUT_FILENO shall be the writable end of the pipe, and the file descriptor *fileno*(*stream*) in the calling process, where *stream* is the stream pointer returned by *popen*( ), shall be the readable end of the pipe.
2. If *mode* is *w*, when the child process is started its file descriptor STDIN_FILENO shall be the readable end of the pipe, and the file descriptor *fileno*(*stream*) in the calling process, where *stream* is the stream pointer returned by *popen*( ), shall be the writable end of the pipe.
3. If *mode* is any other value, the result is undefined.

After *popen*( ), both the parent and the child process shall be capable of executing independently before either terminates.

Pipe streams are byte-oriented.

RETURN VALUE

Upon successful completion, *popen*( ) shall return a pointer to an open stream that can be used to read or write to the pipe. Otherwise, it shall return a null pointer and may set *errno* to indicate the error.

ERRORS

The *popen*( ) function may fail if:

[EMFILE] {FOPEN_MAX} or {STREAM_MAX} streams are currently open in the calling process.

[EINVAL] The *mode* argument is invalid.

The *popen*( ) function may also set *errno* values as described by *fork*( ) or *pipe*( ).

28048 **EXAMPLES**

28049 None.

28050 **APPLICATION USAGE**

28051 Since open files are shared, a mode *r* command can be used as an input filter and a mode *w* |
28052 command as an output filter.

28053 Buffered reading before opening an input filter may leave the standard input of that filter
28054 mispositioned. Similar problems with an output filter may be prevented by careful buffer
28055 flushing; for example, with *fflush*( ).

28056 A stream opened by *popen*( ) should be closed by *pclose*( ).

28057 The behavior of *popen*( ) is specified for values of *mode* of *r* and *w*. Other modes such as *rb* and
28058 *wb* might be supported by specific implementations, but these would not be portable features.
28059 Note that historical implementations of *popen*( ) only check to see if the first character of *mode* is
28060 *r*. Thus, a *mode* of *robert the robot* would be treated as *mode r*, and a *mode* of *anything else* would be
28061 treated as *mode w*.

28062 If the application calls *waitpid*( ) or *waitid*( ) with a *pid* argument greater than 0, and it still has a
28063 stream that was called with *popen*( ) open, it must ensure that *pid* does not refer to the process
28064 started by *popen*( ).

28065 To determine whether or not the environment specified in the Shell and Utilities volume of
28066 IEEE Std 1003.1-200x is present, use the function call:

28067 `sysconf(_SC_2_VERSION)`

28068 (See *sysconf*( )).

28069 **RATIONALE**

28070 The *popen*( ) function should not be used by programs that have set user (or group) ID privileges.
28071 The *fork*( ) and *exec* family of functions (except *execlp*( ) and *execvp*( )), should be used instead.
28072 This prevents any unforeseen manipulation of the environment of the user that could cause
28073 execution of commands not anticipated by the calling program.

28074 If the original and *popen*( )ed processes both intend to read or write or read and write a common
28075 file, and either will be using FILE-type C functions (*fread*( ), *fwrite*( ), and so on), the rules for
28076 sharing file handles must be observed (see Section 2.5.1 (on page 485)).

28077 Since open files are shared, a mode *r* argument can be used as an input filter and a mode *w* |
28078 argument as an output filter.

28079 The behavior of *popen*( ) is specified for modes of *r* and *w*. Other modes such as *rb* and *wb* might
28080 be supported by specific implementations, but these would not be portable features. Note that
28081 historical implementations of *popen*( ) only check to see if the first character of mode is `'r'`.
28082 Thus, a mode of *robert the robot* would be treated as mode *r*, and a mode of *anything else* would be
28083 treated as mode *w*.

28084 If the application calls *waitpid*( ) with a *pid* argument greater than zero, and it still has a
28085 *popen*( )ed stream open, it must ensure that *pid* does not refer to the process started by *popen*( ).

28086 **FUTURE DIRECTIONS**

28087 None.

28088 **SEE ALSO**

28089 *pclose*( ), *pipe*( ), *sysconf*( ), *system*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
28090 **<stdio.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *sh* |

28091 **CHANGE HISTORY**

28092 First released in Issue 1. Derived from Issue 1 of the SVID.

28093 **Issue 5**

28094 A statement is added to the DESCRIPTION indicating that pipe streams are byte-oriented.

28095 **Issue 6**

28096 The following new requirements on POSIX implementations derive from alignment with the
28097 Single UNIX Specification:

28098 • The optional [EMFILE] error condition is added.

28099 **NAME**

28100 posix_fadvise — file advisory information (**ADVANCED REALTIME**)

28101 **SYNOPSIS**

28102 ADV
```
#include <fcntl.h>

int posix_fadvise(int fd, off_t offset, size_t len, int advice);
```
28103

28104

28105 **DESCRIPTION**

28106 The *posix_fadvise*( ) function shall advise the implementation on the expected behavior of the |
28107 application with respect to the data in the file associated with the open file descriptor, *fd*,
28108 starting at *offset* and continuing for *len* bytes. The specified range need not currently exist in the
28109 file. If *len* is zero, all data following *offset* is specified. The implementation may use this
28110 information to optimize handling of the specified data. The *posix_fadvise*( ) function shall have no |
28111 effect on the semantics of other operations on the specified data, although it may affect the |
28112 performance of other operations. |

28113 The advice to be applied to the data is specified by the *advice* parameter and may be one of the
28114 following values:

28115 POSIX_FADV_NORMAL
28116 Specifies that the application has no advice to give on its behavior with respect to the
28117 specified data. It is the default characteristic if no advice is given for an open file.

28118 POSIX_FADV_SEQUENTIAL
28119 Specifies that the application expects to access the specified data sequentially from lower
28120 offsets to higher offsets.

28121 POSIX_FADV_RANDOM
28122 Specifies that the application expects to access the specified data in a random order.

28123 POSIX_FADV_WILLNEED
28124 Specifies that the application expects to access the specified data in the near future.

28125 POSIX_FADV_DONTNEED
28126 Specifies that the application expects that it will not access the specified data in the near
28127 future.

28128 POSIX_FADV_NOREUSE
28129 Specifies that the application expects to access the specified data once and then not reuse it
28130 thereafter.

28131 These values are defined in **<fcntl.h>**. |

28132 **RETURN VALUE**

28133 Upon successful completion, *posix_fadvise*( ) shall return zero; otherwise, an error number shall
28134 be returned to indicate the error.

28135 **ERRORS**

28136 The *posix_fadvise*( ) function shall fail if:

28137 [EBADF] The *fd* argument is not a valid file descriptor.

28138 [EINVAL] The value of *advice* is invalid.

28139 [ESPIPE] The *fd* argument is associated with a pipe or FIFO.

EXAMPLES

None.

APPLICATION USAGE

The *posix_fadvise*( ) function is part of the Advisory Information option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_madvise( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

28154 **NAME**

28155 posix_fallocate — file space control (**ADVANCED REALTIME**)

28156 **SYNOPSIS**

28157 ADV
```
#include <fcntl.h>
```

28158
```
int posix_fallocate(int fd, off_t offset, size_t len);
```

28159

28160 **DESCRIPTION**

28161 The *posix_fallocate*( ) function shall ensure that any required storage for regular file data starting |
28162 at *offset* and continuing for *len* bytes is allocated on the file system storage media. If
28163 *posix_fallocate*( ) returns successfully, subsequent writes to the specified file data shall not fail
28164 due to the lack of free space on the file system storage media.

28165 If the *offset*+*len* is beyond the current file size, then *posix_fallocate*( ) shall adjust the file size to
28166 *offset*+*len*. Otherwise, the file size shall not be changed.

28167 It is implementation-defined whether a previous *posix_fadvise*( ) call influences allocation
28168 strategy.

28169 Space allocated via *posix_fallocate*( ) shall be freed by a successful call to *creat*( ) or *open*( ) that
28170 truncates the size of the file. Space allocated via *posix_fallocate*( ) may be freed by a successful call
28171 to *ftruncate*( ) that reduces the file size to a size smaller than *offset*+*len*.

28172 **RETURN VALUE**

28173 Upon successful completion, *posix_fallocate*( ) shall return zero; otherwise, an error number shall
28174 be returned to indicate the error.

28175 **ERRORS**

28176 The *posix_fallocate*( ) function shall fail if:

28177 [EBADF] The *fd* argument is not a valid file descriptor.

28178 [EBADF] The *fd* argument references a file that was opened without write permission.

28179 [EFBIG] The value of *offset*+*len* is greater than the maximum file size.

28180 [EINTR] A signal was caught during execution.

28181 [EINVAL] The *len* argument was zero or the *offset* argument was less than zero.

28182 [EIO] An I/O error occurred while reading from or writing to a file system.

28183 [ENODEV] The *fd* argument does not refer to a regular file.

28184 [ENOSPC] There is insufficient free space remaining on the file system storage media.

28185 [ESPIPE] The *fd* argument is associated with a pipe or FIFO.

28186 **EXAMPLES**

28187 None.

28188 **APPLICATION USAGE**

28189 The *posix_fallocate*( ) function is part of the Advisory Information option and need not be
28190 provided on all implementations.

28191 **RATIONALE**

28192 None.

28193 **FUTURE DIRECTIONS**

28194 None.

28195 **SEE ALSO**

28196 *creat*( ), *ftruncate*( ), *open*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
28197 **<fcntl.h>**

28198 **CHANGE HISTORY**

28199 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28200 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

28201 **NAME**

28202 posix_madvise — memory advisory information and alignment control (**ADVANCED**
28203 **REALTIME**)

28204 **SYNOPSIS**

28205 ADV
```
#include <sys/mman.h>
```
28206
```
int posix_madvise(void *addr, size_t len, int advice);
```
28207

28208 **DESCRIPTION**

28209 MF|SHM The *posix_madvise*( ) function need only be supported if either the Memory Mapped Files or the
28210 Shared Memory Objects options are supported.

28211 The *posix_madvise*( ) function shall advise the implementation on the expected behavior of the |
28212 application with respect to the data in the memory starting at address *addr*, and continuing for |
28213 *len* bytes. The implementation may use this information to optimize handling of the specified
28214 data. The *posix_madvise*( ) function shall have no effect on the semantics of access to memory in |
28215 the specified range, although it may affect the performance of access. |

28216 The implementation may require that *addr* be a multiple of the page size, which is the value
28217 returned by *sysconf*( ) when the name value _SC_PAGESIZE is used.

28218 The advice to be applied to the memory range is specified by the *advice* parameter and may be
28219 one of the following values:

28220 POSIX_MADV_NORMAL
28221 Specifies that the application has no advice to give on its behavior with respect to the
28222 specified range. It is the default characteristic if no advice is given for a range of memory.

28223 POSIX_MADV_SEQUENTIAL
28224 Specifies that the application expects to access the specified range sequentially from lower
28225 addresses to higher addresses.

28226 POSIX_MADV_RANDOM
28227 Specifies that the application expects to access the specified range in a random order.

28228 POSIX_MADV_WILLNEED
28229 Specifies that the application expects to access the specified range in the near future.

28230 POSIX_MADV_DONTNEED
28231 Specifies that the application expects that it will not access the specified range in the near
28232 future.

28233 These values are defined in the **<sys/mman.h>** header. |

28234 **RETURN VALUE**

28235 Upon successful completion, *posix_madvise*( ) shall return zero; otherwise, an error number shall
28236 be returned to indicate the error.

28237 **ERRORS**

28238 The *posix_madvise*( ) function shall fail if:

28239 [EINVAL] The value of *advice* is invalid.

28240 [ENOMEM] Addresses in the range starting at *addr* and continuing for *len* bytes are partly
28241 or completely outside the range allowed for the address space of the calling
28242 process.

28243 The *posix_madvise*( ) function may fail if:

28244 [EINVAL] The value of *addr* is not a multiple of the value returned by *sysconf*( ) when the
28245 name value _SC_PAGESIZE is used.

28246 [EINVAL] The value of *len* is zero.

28247 **EXAMPLES**

28248 None.

28249 **APPLICATION USAGE**

28250 The *posix_madvise*( ) function is part of the Advisory Information option and need not be
28251 provided on all implementations.

28252 **RATIONALE**

28253 None.

28254 **FUTURE DIRECTIONS**

28255 None.

28256 **SEE ALSO**

28257 *mmap*( ), *posix_fadvise*( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, |
28258 **<sys/mman.h>** |

28259 **CHANGE HISTORY**

28260 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28261 IEEE PASC Interpretation 1003.1 #102 is applied. |

28262 **NAME**

28263 posix_mem_offset — find offset and length of a mapped typed memory block (**ADVANCED**
28264 **REALTIME**)

28265 **SYNOPSIS**

28266 TYM
```
#include <sys/mman.h>
```

28267
```
int posix_mem_offset(const void *restrict addr, size_t len,
    off_t *restrict off, size_t *restrict contig_len,
    int *restrict fildes);
```

28270

28271 **DESCRIPTION**

28272 The *posix_mem_offset*( ) function shall return in the variable pointed to by *off* a value that
28273 identifies the offset (or location), within a memory object, of the memory block currently
28274 mapped at *addr*. The function shall return in the variable pointed to by *fildes*, the descriptor used
28275 (via *mmap*( )) to establish the mapping which contains *addr*. If that descriptor was closed since
28276 the mapping was established, the returned value of *fildes* shall be –1. The *len* argument specifies
28277 the length of the block of the memory object the user wishes the offset for; upon return, the value
28278 pointed to by *contig_len* shall equal either *len*, or the length of the largest contiguous block of the
28279 memory object that is currently mapped to the calling process starting at *addr*, whichever is
28280 smaller.

28281 If the memory object mapped at *addr* is a typed memory object, then if the *off* and *contig_len*
28282 values obtained by calling *posix_mem_offset*( ) are used in a call to *mmap*( ) with a file descriptor
28283 that refers to the same memory pool as *fildes* (either through the same port or through a different
28284 port), and that was opened with neither the POSIX_TYPED_MEM_ALLOCATE nor the
28285 POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, the typed memory area that is mapped shall
28286 be exactly the same area that was mapped at *addr* in the address space of the process that called
28287 *posix_mem_offset*( ).

28288 If the memory object specified by *fildes* is not a typed memory object, then the behavior of this
28289 function is implementation-defined.

28290 **RETURN VALUE**

28291 Upon successful completion, the *posix_mem_offset*( ) function shall return zero; otherwise, the
28292 corresponding error status value shall be returned.

28293 **ERRORS**

28294 The *posix_mem_offset*( ) function shall fail if:

28295 [EACCES] The process has not mapped a memory object supported by this function at
28296 the given address *addr*.

28297 This function shall not return an error code of [EINTR].

28298 **EXAMPLES**

28299 None.

28300 **APPLICATION USAGE**

28301 None.

28302 **RATIONALE**

28303 None.

28304 **FUTURE DIRECTIONS**

28305 None.

28306 **SEE ALSO**

28307 *mmap*( ), *posix_typed_mem_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
28308 **<sys/mman.h>**

28309 **CHANGE HISTORY**

28310 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

28311 **NAME**

28312 posix_memalign — aligned memory allocation (**ADVANCED REALTIME**)

28313 **SYNOPSIS**

28314 ADV
```
#include <stdlib.h>
```

28315
```
int posix_memalign(void **memptr, size_t alignment, size_t size);
```

28316

28317 **DESCRIPTION**

28318 The *posix_memalign*( ) function shall allocate *size* bytes aligned on a boundary specified by
28319 *alignment*, and shall return a pointer to the allocated memory in *memptr*. The value of *alignment*
28320 shall be a multiple of *sizeof*(**void ***), that is also a power of two. Upon successful completion, the
28321 value pointed to by *memptr* shall be a multiple of *alignment*.

28322 CX The *free*( ) function shall deallocate memory that has previously been allocated by
28323 *posix_memalign*( ).

28324 **RETURN VALUE**

28325 Upon successful completion, *posix_memalign*( ) shall return zero; otherwise, an error number
28326 shall be returned to indicate the error.

28327 **ERRORS**

28328 The *posix_memalign*( ) function shall fail if:

28329 [EINVAL] The value of the alignment parameter is not a power of two multiple of
28330 *sizeof*(**void ***).

28331 [ENOMEM] There is insufficient memory available with the requested alignment.

28332 **EXAMPLES**

28333 None.

28334 **APPLICATION USAGE**

28335 The *posix_memalign*( ) function is part of the Advisory Information option and need not be
28336 provided on all implementations.

28337 **RATIONALE**

28338 None.

28339 **FUTURE DIRECTIONS**

28340 None.

28341 **SEE ALSO**

28342 *free*( ), *malloc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

28343 **CHANGE HISTORY**

28344 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28345 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

posix_openpt — open a pseudo terminal device

SYNOPSIS

XSI
```
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt(int oflag);
```

DESCRIPTION

The *posix_openpt*( ) function shall establish a connection between a master device for a pseudo-terminal and a file descriptor. The file descriptor is used by other I/O functions that refer to that pseudo-terminal.

The file status flags and file access modes of the open file description shall be set according to the value of *oflag*.

Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**:

O_RDWR — Open for reading and writing.

O_NOCTTY — If set *posix_openpt*( ) shall not cause the terminal device to become the controlling terminal for the process.

The behavior of other values for the *oflag* argument is unspecified.

RETURN VALUE

Upon successful completion, the *posix_openpt*( ) function shall open a master pseudo-terminal device and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *posix_openpt*( ) function shall fail if:

[EMFILE] — {OPEN_MAX} file descriptors are currently open in the calling process.

[ENFILE] — The maximum allowable number of files is currently open in the system.

The *posix_openpt*( ) function may fail if:

[EINVAL] — The value of *oflag* is not valid.

[EAGAIN] — Out of pseudo-terminal resources.

XSR [ENOSR] — Out of STREAMS resources.

EXAMPLES

Opening a Pseudo-Terminal and Returning the Name of the Slave Device and a File Descriptor

```
#include <fcntl.h>
#include <stdio.h>

int masterfd, slavefd;
char *slavedevice;

masterfd = posix_openpt(O_RDWR|O_NOCTTY);

if (masterfd == -1
    || grantpt (masterfd) == -1
```

```
28387           || unlockpt (masterfd) == -1
28388           || (slavedevice = ptsname (masterfd)) == NULL)
28389           return -1;
28390       printf("slave device is: %s\n", slavedevice);
28391       slavefd = open(slave, O_RDWR|O_NOCTTY);
28392       if (slavefd < 0)
28393           return -1;
```

28394 **APPLICATION USAGE**

28395 This function is a method for portably obtaining a file descriptor of a master terminal device for
28396 a pseudo-terminal. The *grantpt*( ) and *ptsname*( ) functions can be used to manipulate mode and
28397 ownership permissions, and to obtain the name of the slave device, respectively.

28398 **RATIONALE**

28399 The standard developers considered the matter of adding a special device for cloning master
28400 pseudo-terminals: the **/dev/ptmx** device. However, consensus could not be reached, and it was
28401 felt that adding a new function would permit other implementations. The *posix_openpt*( )
28402 function is designed to complement the *grantpt*( ), *ptsname*( ), and *unlockpt*( ) functions.

28403 On implementations supporting the **/dev/ptmx** clone device, opening the master device of a
28404 pseudo-terminal is simply:

```
28405       mfdp = open("/dev/ptmx", oflag );
28406       if (mfdp < 0)
28407           return -1;
```

28408 **FUTURE DIRECTIONS**

28409 None.

28410 **SEE ALSO**

28411 *grantpt*( ), *open*( ), *ptsname*( ), *unlockpt*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
28412 **<fcntl.h>**

28413 **CHANGE HISTORY**

28414 First released in Issue 6.

NAME

posix_spawn, posix_spawnp — spawn a process (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawn(pid_t *restrict pid, const char *restrict path,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp,
    char *const argv[restrict], char *const envp[restrict]);
int posix_spawnp(pid_t *restrict pid, const char *restrict file,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp,
    char *const argv[restrict], char * const envp[restrict]);
```

DESCRIPTION

The *posix_spawn*( ) and *posix_spawnp*( ) functions shall create a new process (child process) from the specified process image. The new process image shall be constructed from a regular executable file called the new process image file.

When a C program is executed as the result of this call, it shall be entered as a C language function call as follows:

```
int main(int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

shall be initialized as a pointer to an array of character pointers to the environment strings.

The argument *argv* is an array of character pointers to null-terminated strings. The last member of this array shall be a null pointer and is not counted in *argc*. These strings constitute the argument list available to the new process image. The value in *argv*[0] should point to a filename that is associated with the process image being started by the *posix_spawn*( ) or *posix_spawnp*( ) function.

The argument *envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated by a null pointer.

The number of bytes available for the child process' combined argument and environment lists is {ARG_MAX}. The implementation shall specify in the system documentation (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance) whether any list overhead, such as length words, null terminators, pointers, or alignment bytes, is included in this total.

The *path* argument to *posix_spawn*( ) is a pathname that identifies the new process image file to execute.

The *file* parameter to *posix_spawnp*( ) shall be used to construct a pathname that identifies the new process image file. If the *file* parameter contains a slash character, the *file* parameter shall be used as the pathname for the new process image file. Otherwise, the path prefix for this file shall be obtained by a search of the directories passed as the environment variable *PATH* (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables). If this environment variable is not defined, the results of the search are implementation-defined.

If *file_actions* is a null pointer, then file descriptors open in the calling process shall remain open in the child process, except for those whose close-on-*exec* flag FD_CLOEXEC is set (see *fcntl*( )). For those file descriptors that remain open, all attributes of the corresponding open file descriptions, including file locks (see *fcntl*( )), shall remain unchanged.

If *file_actions* is not NULL, then the file descriptors open in the child process shall be those open in the calling process as modified by the spawn file actions object pointed to by *file_actions* and the FD_CLOEXEC flag of each remaining open file descriptor after the spawn file actions have been processed. The effective order of processing the spawn file actions shall be:

1. The set of open file descriptors for the child process shall initially be the same set as is open for the calling process. All attributes of the corresponding open file descriptions, including file locks (see *fcntl*( )), shall remain unchanged.
2. The signal mask, signal default actions, and the effective user and group IDs for the child process shall be changed as specified in the attributes object referenced by *attrp*.
3. The file actions specified by the spawn file actions object shall be performed in the order in which they were added to the spawn file actions object.
4. Any file descriptor that has its FD_CLOEXEC flag set (see *fcntl*( )) shall be closed.

The **posix_spawnattr_t** spawn attributes object type is defined in **<spawn.h>**. It shall contain at least the attributes defined below.

If the POSIX_SPAWN_SETPGROUP flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, and the *spawn-pgroup* attribute of the same object is non-zero, then the child's process group shall be as specified in the *spawn-pgroup* attribute of the object referenced by *attrp*.

As a special case, if the POSIX_SPAWN_SETPGROUP flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, and the *spawn-pgroup* attribute of the same object is set to zero, then the child shall be in a new process group with a process group ID equal to its process ID.

If the POSIX_SPAWN_SETPGROUP flag is not set in the *spawn-flags* attribute of the object referenced by *attrp*, the new child process shall inherit the parent's process group.

PS If the POSIX_SPAWN_SETSCHEDPARAM flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, but POSIX_SPAWN_SETSCHEDULER is not set, the new process image shall initially have the scheduling policy of the calling process with the scheduling parameters specified in the *spawn-schedparam* attribute of the object referenced by *attrp*.

If the POSIX_SPAWN_SETSCHEDULER flag is set in *spawn-flags* attribute of the object referenced by *attrp* (regardless of the setting of the POSIX_SPAWN_SETSCHEDPARAM flag), the new process image shall initially have the scheduling policy specified in the *spawn-schedpolicy* attribute of the object referenced by *attrp* and the scheduling parameters specified in the *spawn-schedparam* attribute of the same object.

The POSIX_SPAWN_RESETIDS flag in the *spawn-flags* attribute of the object referenced by *attrp* governs the effective user ID of the child process. If this flag is not set, the child process shall inherit the parent process' effective user ID. If this flag is set, the child process' effective user ID shall be reset to the parent's real user ID. In either case, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the child process shall become that file's owner ID before the new process image begins execution.

The POSIX_SPAWN_RESETIDS flag in the *spawn-flags* attribute of the object referenced by *attrp* also governs the effective group ID of the child process. If this flag is not set, the child process shall inherit the parent process' effective group ID. If this flag is set, the child process' effective group ID shall be reset to the parent's real group ID. In either case, if the set-group-ID mode bit

of the new process image file is set, the effective group ID of the child process shall become that file's group ID before the new process image begins execution.

If the POSIX_SPAWN_SETSIGMASK flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, the child process shall initially have the signal mask specified in the *spawn-sigmask* attribute of the object referenced by *attrp*.

If the POSIX_SPAWN_SETSIGDEF flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, the signals specified in the *spawn-sigdefault* attribute of the same object shall be set to their default actions in the child process. Signals set to the default action in the parent process shall be set to the default action in the child process.

Signals set to be caught by the calling process shall be set to the default action in the child process.

Except for SIGCHLD, signals set to be ignored by the calling process image shall be set to be ignored by the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the *spawn-flags* attribute of the object referenced by *attrp* and the signals being indicated in the *spawn-sigdefault* attribute of the object referenced by *attrp*.

If the SIGCHLD signal is set to be ignored by the calling process, it is unspecified whether the SIGCHLD signal is set to be ignored or to the default action in the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the *spawn_flags* attribute of the object referenced by *attrp* and the SIGCHLD signal being indicated in the *spawn_sigdefault* attribute of the object referenced by *attrp*.

If the value of the *attrp* pointer is NULL, then the default values are used.

All process attributes, other than those influenced by the attributes set in the object referenced by *attrp* as specified above or by the file descriptor manipulations specified in *file_actions*, shall appear in the new process image as though *fork*( ) had been called to create a child process and then a member of the *exec* family of functions had been called by the child process to execute the new process image.

THR It is implementation-defined whether the fork handlers are run when *posix_spawn*( ) or *posix_spawnp*( ) is called.

RETURN VALUE

Upon successful completion, *posix_spawn*( ) and *posix_spawnp*( ) shall return the process ID of the child process to the parent process, in the variable pointed to by a non-NULL *pid* argument, and shall return zero as the function return value. Otherwise, no child process shall be created, the value stored into the variable pointed to by a non-NULL *pid* is unspecified, and an error number shall be returned as the function return value to indicate the error. If the *pid* argument is a null pointer, the process ID of the child is not returned to the caller.

ERRORS

The *posix_spawn*( ) and *posix_spawnp*( ) functions may fail if:

[EINVAL] The value specified by *file_actions* or *attrp* is invalid.

If this error occurs after the calling process successfully returns from the *posix_spawn*( ) or *posix_spawnp*( ) function, the child process may exit with exit status 127.

If *posix_spawn*( ) or *posix_spawnp*( ) fail for any of the reasons that would cause *fork*( ) or one of the *exec* family of functions to fail, an error value shall be returned as described by *fork*( ) and *exec*, respectively (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

If POSIX_SPAWN_SETPGROUP is set in the *spawn-flags* attribute of the object referenced by *attrp*, and *posix_spawn*( ) or *posix_spawnp*( ) fails while changing the child's process group, an error value shall be returned as described by *setpgid*( ) (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

PS If POSIX_SPAWN_SETSCHEDPARAM is set and POSIX_SPAWN_SETSCHEDULER is not set in the *spawn-flags* attribute of the object referenced by *attrp*, then if *posix_spawn*( ) or *posix_spawnp*( ) fails for any of the reasons that would cause *sched_setparam*( ) to fail, an error value shall be returned as described by *sched_setparam*( ) (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

If POSIX_SPAWN_SETSCHEDULER is set in the *spawn-flags* attribute of the object referenced by *attrp*, and if *posix_spawn*( ) or *posix_spawnp*( ) fails for any of the reasons that would cause *sched_setscheduler*( ) to fail, an error value shall be returned as described by *sched_setscheduler*( ) (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

If the *file_actions* argument is not NULL, and specifies any *close*, *dup2*, or *open* actions to be performed, and if *posix_spawn*( ) or *posix_spawnp*( ) fails for any of the reasons that would cause *close*( ), *dup2*( ), or *open*( ) to fail, an error value shall be returned as described by *close*( ), *dup2*( ), and *open*( ), respectively (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127). An open file action may, by itself, result in any of the errors described by *close*( ) or *dup2*( ), in addition to those described by *open*( ).

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

The *posix_spawn*( ) function and its close relation *posix_spawnp*( ) have been introduced to overcome the following perceived difficulties with *fork*( ): the *fork*( ) function is difficult or impossible to implement without swapping or dynamic address translation.

- Swapping is generally too slow for a realtime environment.
- Dynamic address translation is not available everywhere that POSIX might be useful.
- Processes are too useful to simply option out of POSIX whenever it must run without address translation or other MMU services,

Thus, POSIX needs process creation and file execution primitives that can be efficiently implemented without address translation or other MMU services.

The *posix_spawn*( ) function is implementable as a library routine, but both *posix_spawn*( ) and *posix_spawnp*( ) are designed as kernel operations. Also, although they may be an efficient replacement for many *fork*( )/*exec* pairs, their goal is to provide useful process creation primitives for systems that have difficulty with *fork*( ), not to provide drop-in replacements for *fork*( )/*exec*.

This view of the role of *posix_spawn*( ) and *posix_spawnp*( ) influenced the design of their API. It does not attempt to provide the full functionality of *fork*( )/*exec* in which arbitrary user-specified operations of any sort are permitted between the creation of the child process and the execution of the new process image; any attempt to reach that level would need to provide a programming language as parameters. Instead, *posix_spawn*( ) and *posix_spawnp*( ) are process creation primitives like the *Start_Process* and *Start_Process_Search* Ada language bindings package *POSIX_Process_Primitives* and also like those in many operating systems that are not UNIX

systems, but with some POSIX-specific additions.

To achieve its coverage goals, *posix_spawn*( ) and *posix_spawnp*( ) have control of six types of inheritance: file descriptors, process group ID, user and group ID, signal mask, scheduling, and whether each signal ignored in the parent will remain ignored in the child, or be reset to its default action in the child.

Control of file descriptors is required to allow an independently written child process image to access data streams opened by and even generated or read by the parent process without being specifically coded to know which parent files and file descriptors are to be used. Control of the process group ID is required to control how the child process' job control relates to that of the parent.

Control of the signal mask and signal defaulting is sufficient to support the implementation of *system*( ). Although support for *system*( ) is not explicitly one of the goals for *posix_spawn*( ) and *posix_spawnp*( ), it is covered under the ''at least 50%'' coverage goal.

The intention is that the normal file descriptor inheritance across *fork*( ), the subsequent effect of the specified spawn file actions, and the normal file descriptor inheritance across one of the *exec* family of functions should fully specify open file inheritance. The implementation need make no decisions regarding the set of open file descriptors when the child process image begins execution, those decisions having already been made by the caller and expressed as the set of open file descriptors and their FD_CLOEXEC flags at the time of the call and the spawn file actions object specified in the call. We have been assured that in cases where the POSIX *Start_Process* Ada primitives have been implemented in a library, this method of controlling file descriptor inheritance may be implemented very easily.

We can identify several problems with *posix_spawn*( ) and *posix_spawnp*( ), but there does not appear to be a solution that introduces fewer problems. Environment modification for child process attributes not specifiable via the *attrp* or *file_actions* arguments must be done in the parent process, and since the parent generally wants to save its context, it is more costly than similar functionality with *fork*( )/*exec*. It is also complicated to modify the environment of a multi-threaded process temporarily, since all threads must agree when it is safe for the environment to be changed. However, this cost is only borne by those invocations of *posix_spawn*( ) and *posix_spawnp*( ) that use the additional functionality. Since extensive modifications are not the usual case, and are particularly unlikely in time-critical code, keeping much of the environment control out of *posix_spawn*( ) and *posix_spawnp*( ) is appropriate design.

The *posix_spawn*( ) and *posix_spawnp*( ) functions do not have all the power of *fork*( )/*exec*. This is to be expected. The *fork*( ) function is a wonderfully powerful operation. We do not expect to duplicate its functionality in a simple, fast function with no special hardware requirements. It is worth noting that *posix_spawn*( ) and *posix_spawnp*( ) are very similar to the process creation operations on many operating systems that are not UNIX systems.

Requirements

The requirements for *posix_spawn*( ) and *posix_spawnp*( ) are:

- They must be implementable without an MMU or unusual hardware.
- They must be compatible with existing POSIX standards.

Additional goals are:

- They should be efficiently implementable.
- They should be able to replace at least 50% of typical executions of *fork*( ).

- A system with *posix_spawn*( ) and *posix_spawnp*( ) and without *fork*( ) should be useful, at least for realtime applications.
- A system with *fork*( ) and the *exec* family should be able to implement *posix_spawn*( ) and *posix_spawnp*( ) as library routines.

Two-Syntax

POSIX *exec* has several calling sequences with approximately the same functionality. These appear to be required for compatibility with existing practice. Since the existing practice for the *posix_spawn**( ) functions is otherwise substantially unlike POSIX, we feel that simplicity outweighs compatibility. There are, therefore, only two names for the *posix_spawn**( ) functions.

The parameter list does not differ between *posix_spawn*( ) and *posix_spawnp*( ); *posix_spawnp*( ) interprets the second parameter more elaborately than *posix_spawn*( ).

Compatibility with POSIX.5 (Ada)

The *Start_Process* and *Start_Process_Search* procedures from the POSIX_Process_Primitives package from the Ada language binding to POSIX.1 encapsulate *fork*( ) and *exec* functionality in a manner similar to that of *posix_spawn*( ) and *posix_spawnp*( ). Originally, in keeping with our simplicity goal, the standard developers had limited the capabilities of *posix_spawn*( ) and *posix_spawnp*( ) to a subset of the capabilities of *Start_Process* and *Start_Process_Search*; certain non-default capabilities were not supported. However, based on suggestions by the ballot group to improve file descriptor mapping or drop it, and on the advice of an Ada Language Bindings working group member, the standard developers decided that *posix_spawn*( ) and *posix_spawnp*( ) should be sufficiently powerful to implement *Start_Process* and *Start_Process_Search*. The rationale is that if the Ada language binding to such a primitive had already been approved as an IEEE standard, there can be little justification for not approving the functionally-equivalent parts of a C binding. The only three capabilities provided by *posix_spawn*( ) and *posix_spawnp*( ) that are not provided by *Start_Process* and *Start_Process_Search* are optionally specifying the child's process group ID, the set of signals to be reset to default signal handling in the child process, and the child's scheduling policy and parameters.

For the Ada language binding for *Start_Process* to be implemented with *posix_spawn*( ), that binding would need to explicitly pass an empty signal mask and the parent's environment to *posix_spawn*( ) whenever the caller of *Start_Process* allowed these arguments to default, since *posix_spawn*( ) does not provide such defaults. The ability of *Start_Process* to mask user-specified signals during its execution is functionally unique to the Ada language binding and must be dealt with in the binding separately from the call to *posix_spawn*( ).

Process Group

The process group inheritance field can be used to join the child process with an existing process group. By assigning a value of zero to the *spawn-pgroup* attribute of the object referenced by *attrp*, the *setpgid*( ) mechanism will place the child process in a new process group.

28677 **Threads**

28678 Without the *posix_spawn*( ) and *posix_spawnp*( ) functions, systems without address translation
28679 can still use threads to give an abstraction of concurrency. In many cases, thread creation
28680 suffices, but it is not always a good substitute. The *posix_spawn*( ) and *posix_spawnp*( ) functions
28681 are considerably ‘‘heavier’’ than thread creation. Processes have several important attributes that
28682 threads do not. Even without address translation, a process may have base-and-bound memory
28683 protection. Each process has a process environment including security attributes and file
28684 capabilities, and powerful scheduling attributes. Processes abstract the behavior of non-
28685 uniform-memory-architecture multi-processors better than threads, and they are more
28686 convenient to use for activities that are not closely linked.

28687 The *posix_spawn*( ) and *posix_spawnp*( ) functions may not bring support for multiple processes to
28688 every configuration. Process creation is not the only piece of operating system support required
28689 to support multiple processes. The total cost of support for multiple processes may be quite high
28690 in some circumstances. Existing practice shows that support for multiple processes is
28691 uncommon and threads are common among ‘‘tiny kernels’’. There should, therefore, probably
28692 continue to be AEPs for operating systems with only one process.

28693 **Asynchronous Error Notification**

28694 A library implementation of *posix_spawn*( ) or *posix_spawnp*( ) may not be able to detect all
28695 possible errors before it forks the child process. IEEE Std 1003.1-200x provides for an error
28696 indication returned from a child process which could not successfully complete the spawn
28697 operation via a special exit status which may be detected using the status value returned by
28698 *wait*( ) and *waitpid*( ).

28699 The *stat_val* interface and the macros used to interpret it are not well suited to the purpose of
28700 returning API errors, but they are the only path available to a library implementation. Thus, an
28701 implementation may cause the child process to exit with exit status 127 for any error detected
28702 during the spawn process after the *posix_spawn*( ) or *posix_spawnp*( ) function has successfully
28703 returned.

28704 The standard developers had proposed using two additional macros to interpret *stat_val*. The
28705 first, WIFSPAWNFAIL, would have detected a status that indicated that the child exited because
28706 of an error detected during the *posix_spawn*( ) or *posix_spawnp*( ) operations rather than during
28707 actual execution of the child process image; the second, WSPAWNERRNO, would have
28708 extracted the error value if WIFSPAWNFAIL indicated a failure. Unfortunately, the ballot group
28709 strongly opposed this because it would make a library implementation of *posix_spawn*( ) or
28710 *posix_spawnp*( ) dependent on kernel modifications to *waitpid*( ) to be able to embed special
28711 information in *stat_val* to indicate a spawn failure.

28712 The 8 bits of child process exit status that are guaranteed by IEEE Std 1003.1-200x to be
28713 accessible to the waiting parent process are insufficient to disambiguate a spawn error from any
28714 other kind of error that may be returned by an arbitrary process image. No other bits of the exit
28715 status are required to be visible in *stat_val*, so these macros could not be strictly implemented at
28716 the library level. Reserving an exit status of 127 for such spawn errors is consistent with the use
28717 of this value by *system*( ) and *popen*( ) to signal failures in these operations that occur after the
28718 function has returned but before a shell is able to execute. The exit status of 127 does not
28719 uniquely identify this class of error, nor does it provide any detailed information on the nature
28720 of the failure. Note that a kernel implementation of *posix_spawn*( ) or *posix_spawnp*( ) is permitted
28721 (and encouraged) to return any possible error as the function value, thus providing more
28722 detailed failure information to the parent process.

28723 Thus, no special macros are available to isolate asynchronous *posix_spawn*( ) or *posix_spawnp*( )
28724 errors. Instead, errors detected by the *posix_spawn*( ) or *posix_spawnp*( ) operations in the context

of the child process before the new process image executes are reported by setting the child's exit status to 127. The calling process may use the WIFEXITED and WEXITSTATUS macros on the *stat_val* stored by the *wait*( ) or *waitpid*( ) functions to detect spawn failures to the extent that other status values with which the child process image may exit (before the parent can conclusively determine that the child process image has begun execution) are distinct from exit status 127.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *chmod*( ), *close*( ), *dup*( ), *exec*, *exit*( ), *fcntl*( ), *fork*( ), *kill*( ), *open*( ), *posix_spawn_file_actions_addclose*( ), *posix_spawn_file_actions_adddup2*( ), *posix_spawn_file_actions_addopen*( ), *posix_spawn_file_actions_destroy*( ), *posix_spawn_file_actions_init*( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getsigdefault*( ), *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ), *sched_setparam*( ), *sched_setscheduler*( ), *setpgid*( ), *setuid*( ), *stat*( ), *times*( ), *wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #103 is applied, noting that the signal default actions are changed as well as the signal mask in step 2. |

IEEE PASC Interpretation 1003.1 #132 is applied. |

28749 **NAME**

28750 posix_spawn_file_actions_addclose, posix_spawn_file_actions_addopen — add close or open
28751 action to spawn file actions object (**ADVANCED REALTIME**)

28752 **SYNOPSIS**

28753 SPN
```
#include <spawn.h>

int posix_spawn_file_actions_addclose(posix_spawn_file_actions_t *
    file_actions, int fildes);
int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *restrict
    file_actions, int fildes, const char *restrict path,
    int oflag, mode_t mode);
```
28759

28760 **DESCRIPTION**

28761 These functions shall add or delete a close or open action to a spawn file actions object. |

28762 A spawn file actions object is of type **posix_spawn_file_actions_t** (defined in **<spawn.h>**) and is |
28763 used to specify a series of actions to be performed by a *posix_spawn*( ) or *posix_spawnp*( )
28764 operation in order to arrive at the set of open file descriptors for the child process given the set of
28765 open file descriptors of the parent. IEEE Std 1003.1-200x does not define comparison or
28766 assignment operators for the type **posix_spawn_file_actions_t**.

28767 A spawn file actions object, when passed to *posix_spawn*( ) or *posix_spawnp*( ), shall specify how
28768 the set of open file descriptors in the calling process is transformed into a set of potentially open
28769 file descriptors for the spawned process. This transformation shall be as if the specified sequence
28770 of actions was performed exactly once, in the context of the spawned process (prior to execution
28771 of the new process image), in the order in which the actions were added to the object;
28772 additionally, when the new process image is executed, any file descriptor (from this new set)
28773 which has its FD_CLOEXEC flag set shall be closed (see *posix_spawn*( )). |

28774 The *posix_spawn_file_actions_addclose*( ) function shall add a *close* action to the object referenced |
28775 by *file_actions* that shall cause the file descriptor *fildes* to be closed (as if *close*(*fildes*) had been |
28776 called) when a new process is spawned using this file actions object.

28777 The *posix_spawn_file_actions_addopen*( ) function shall add an *open* action to the object referenced |
28778 by *file_actions* that shall cause the file named by *path* to be opened (as if *open*(*path*, *oflag*, *mode*) |
28779 had been called, and the returned file descriptor, if not *fildes*, had been changed to *fildes*) when a
28780 new process is spawned using this file actions object. If *fildes* was already an open file descriptor,
28781 it shall be closed before the new file is opened.

28782 The string described by *path* shall be copied by the *posix_spawn_file_actions_addopen*( ) function. |

28783 **RETURN VALUE**

28784 Upon successful completion, these functions shall return zero; otherwise, an error number shall
28785 be returned to indicate the error.

28786 **ERRORS**

28787 These functions shall fail if:

28788 [EBADF] The value specified by *fildes* is negative or greater than or equal to
28789 {OPEN_MAX}.

28790 These functions may fail if:

28791 [EINVAL] The value specified by *file_actions* is invalid.

28792 [ENOMEM] Insufficient memory exists to add to the spawn file actions object.

It shall not be considered an error for the *fildes* argument passed to these functions to specify a file descriptor for which the specified operation could not be performed at the time of the call. Any such error will be detected when the associated file actions object is later used during a *posix_spawn*( ) or *posix_spawnp*( ) operation.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

A spawn file actions object may be initialized to contain an ordered sequence of *close*( ), *dup2*( ), and *open*( ) operations to be used by *posix_spawn*( ) or *posix_spawnp*( ) to arrive at the set of open file descriptors inherited by the spawned process from the set of open file descriptors in the parent at the time of the *posix_spawn*( ) or *posix_spawnp*( ) call. It had been suggested that the *close*( ) and *dup2*( ) operations alone are sufficient to rearrange file descriptors, and that files which need to be opened for use by the spawned process can be handled either by having the calling process open them before the *posix_spawn*( ) or *posix_spawnp*( ) call (and close them after), or by passing filenames to the spawned process (in *argv*) so that it may open them itself. The standard developers recommend that applications use one of these two methods when practical, since detailed error status on a failed open operation is always available to the application this way. However, the standard developers feel that allowing a spawn file actions object to specify open operations is still appropriate because:

1. It is consistent with equivalent POSIX.5 (Ada) functionality.
2. It supports the I/O redirection paradigm commonly employed by POSIX programs designed to be invoked from a shell. When such a program is the child process, it may not be designed to open files on its own.
3. It allows file opens that might otherwise fail or violate file ownership/access rights if executed by the parent process.

Regarding 2. above, note that the spawn open file action provides to *posix_spawn*( ) and *posix_spawnp*( ) the same capability that the shell redirection operators provide to *system*( ), only without the intervening execution of a shell; for example:

```
system ("myprog <file1 3<file2");
```

Regarding 3. above, note that if the calling process needs to open one or more files for access by the spawned process, but has insufficient spare file descriptors, then the open action is necessary to allow the *open*( ) to occur in the context of the child process after other file descriptors have been closed (that must remain open in the parent).

Additionally, if a parent is executed from a file having a ‘‘set-user-id’’ mode bit set and the POSIX_SPAWN_RESETIDS flag is set in the spawn attributes, a file created within the parent process will (possibly incorrectly) have the parent’s effective user ID as its owner, whereas a file created via an *open*( ) action during *posix_spawn*( ) or *posix_spawnp*( ) will have the parent’s real ID as its owner; and an open by the parent process may successfully open a file to which the real user should not have access or fail to open a file to which the real user should have access.

File Descriptor Mapping

The standard developers had originally proposed using an array which specified the mapping of child file descriptors back to those of the parent. It was pointed out by the ballot group that it is not possible to reshuffle file descriptors arbitrarily in a library implementation of *posix_spawn*( ) or *posix_spawnp*( ) without provision for one or more spare file descriptor entries (which simply may not be available). Such an array requires that an implementation develop a complex strategy to achieve the desired mapping without inadvertently closing the wrong file descriptor at the wrong time.

It was noted by a member of the Ada Language Bindings working group that the approved Ada Language *Start_Process* family of POSIX process primitives use a caller-specified set of file actions to alter the normal *fork*( )/*exec* semantics for inheritance of file descriptors in a very flexible way, yet no such problems exist because the burden of determining how to achieve the final file descriptor mapping is completely on the application. Furthermore, although the file actions interface appears frightening at first glance, it is actually quite simple to implement in either a library or the kernel.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *dup*( ), *open*( ), *posix_spawn*( ), *posix_spawn_file_actions_adddup2*( ), *posix_spawn_file_actions_destroy*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #105 is applied, adding a note to the DESCRIPTION that the string pointed to by *path* is copied by the *posix_spawn_file_actions_addopen*( ) function.

28859 **NAME**

28860 posix_spawn_file_actions_adddup2 — add dup2 action to spawn file actions object
28861 (**ADVANCED REALTIME**)

28862 **SYNOPSIS**

28863 SPN
```
#include <spawn.h>

int posix_spawn_file_actions_adddup2(posix_spawn_file_actions_t *
    file_actions, int fildes, int newfildes);
```
28866

28867 **DESCRIPTION**

28868 The *posix_spawn_file_actions_adddup2*( ) function shall add a *dup2*( ) action to the object |
28869 referenced by *file_actions* that shall cause the file descriptor *fildes* to be duplicated as *newfildes* (as |
28870 if *dup2*(*fildes*, *newfildes*) had been called) when a new process is spawned using this file actions
28871 object. |

28872 A spawn file actions object is as defined in *posix_spawn_file_actions_addclose*( ). |

28873 **RETURN VALUE**

28874 Upon successful completion, the *posix_spawn_file_actions_adddup2*( ) function shall return zero;
28875 otherwise, an error number shall be returned to indicate the error.

28876 **ERRORS**

28877 The *posix_spawn_file_actions_adddup2*( ) function shall fail if:

28878 [EBADF] The value specified by *fildes* or *newfildes* is negative or greater than or equal to
28879 {OPEN_MAX}.

28880 [ENOMEM] Insufficient memory exists to add to the spawn file actions object.

28881 The *posix_spawn_file_actions_adddup2*( ) function may fail if:

28882 [EINVAL] The value specified by *file_actions* is invalid.

28883 It shall not be considered an error for the *fildes* argument passed to the
28884 *posix_spawn_file_actions_adddup2*( ) function to specify a file descriptor for which the specified
28885 operation could not be performed at the time of the call. Any such error will be detected when
28886 the associated file actions object is later used during a *posix_spawn*( ) or *posix_spawnp*( )
28887 operation.

28888 **EXAMPLES**

28889 None.

28890 **APPLICATION USAGE**

28891 The *posix_spawn_file_actions_adddup2*( ) function is part of the Spawn option and need not be
28892 provided on all implementations.

28893 **RATIONALE**

28894 Refer to the RATIONALE in *posix_spawn_file_actions_addclose*( ).

28895 **FUTURE DIRECTIONS**

28896 None.

28897 **SEE ALSO**

28898 *dup*( ), *posix_spawn*( ), *posix_spawn_file_actions_addclose*( ), *posix_spawn_file_actions_destroy*( ),
28899 *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

28900 **CHANGE HISTORY**

28901 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28902 IEEE PASC Interpretation 1003.1 #104 is applied, noting that the [EBADF] error can apply to the |
28903 *newfildes* argument in addition to *fildes*.

28904 **NAME**

28905 posix_spawn_file_actions_addopen — add open action to spawn file actions object
28906 (**ADVANCED REALTIME**)

28907 **SYNOPSIS**

28908 SPN
```
#include <spawn.h>
```

28909
```
int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *restrict
    file_actions, int fildes, const char *restrict path,
    int oflag, mode_t mode);
```

28912

28913 **DESCRIPTION**

28914 Refer to *posix_spawn_file_actions_addclose*( ).

28915 **NAME**

28916 posix_spawn_file_actions_destroy, posix_spawn_file_actions_init — destroy and initialize
28917 spawn file actions object (**ADVANCED REALTIME**)

28918 **SYNOPSIS**

28919 SPN
```
#include <spawn.h>

int posix_spawn_file_actions_destroy(posix_spawn_file_actions_t *
    file_actions);
int posix_spawn_file_actions_init(posix_spawn_file_actions_t *
    file_actions);
```
28924

28925 **DESCRIPTION**

28926 The *posix_spawn_file_actions_destroy*( ) function shall destroy the object referenced by *file_actions*; |
28927 the object becomes, in effect, uninitialized. An implementation may cause
28928 *posix_spawn_file_actions_destroy*( ) to set the object referenced by *file_actions* to an invalid value. A
28929 destroyed spawn file actions object can be reinitialized using *posix_spawn_file_actions_init*( ); the
28930 results of otherwise referencing the object after it has been destroyed are undefined.

28931 The *posix_spawn_file_actions_init*( ) function shall initialize the object referenced by *file_actions* to
28932 contain no file actions for *posix_spawn*( ) or *posix_spawnp*( ) to perform.

28933 A spawn file actions object is as defined in *posix_spawn_file_actions_addclose*( ). |

28934 The effect of initializing an already initialized spawn file actions object is undefined. |

28935 **RETURN VALUE**

28936 Upon successful completion, these functions shall return zero; otherwise, an error number shall
28937 be returned to indicate the error.

28938 **ERRORS**

28939 The *posix_spawn_file_actions_init*( ) function shall fail if:

28940 [ENOMEM] Insufficient memory exists to initialize the spawn file actions object.

28941 The *posix_spawn_file_actions_destroy*( ) function may fail if:

28942 [EINVAL] The value specified by *file_actions* is invalid.

28943 **EXAMPLES**

28944 None.

28945 **APPLICATION USAGE**

28946 These functions are part of the Spawn option and need not be provided on all implementations.

28947 **RATIONALE**

28948 Refer to the RATIONALE in *posix_spawn_file_actions_addclose*( ).

28949 **FUTURE DIRECTIONS**

28950 None.

28951 **SEE ALSO**

28952 *posix_spawn*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

28953 **CHANGE HISTORY**

28954 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28955 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

28956 **NAME**

28957 posix_spawn_file_actions_init — initialize spawn file actions object (**ADVANCED REALTIME**)

28958 **SYNOPSIS**

28959 SPN
```
#include <spawn.h>

int posix_spawn_file_actions_init(posix_spawn_file_actions_t *
    file_actions);
```
28960

28961

28962

28963 **DESCRIPTION**

28964 Refer to *posix_spawn_file_actions_destroy*( ).

NAME

posix_spawnattr_destroy, posix_spawnattr_init — destroy and initialize spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawnattr_destroy(posix_spawnattr_t *attr);
int posix_spawnattr_init(posix_spawnattr_t *attr);
```

DESCRIPTION

The *posix_spawnattr_destroy*( ) function shall destroy a spawn attributes object. A destroyed *attr* attributes object can be reinitialized using *posix_spawnattr_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined. An implementation may cause *posix_spawnattr_destroy*( ) to set the object referenced by *attr* to an invalid value.

The *posix_spawnattr_init*( ) function shall initialize a spawn attributes object *attr* with the default value for all of the individual attributes used by the implementation. Results are undefined if *posix_spawnattr_init*( ) is called specifying an already initialized *attr* attributes object.

A spawn attributes object is of type **posix_spawnattr_t** (defined in **<spawn.h>**) and is used to specify the inheritance of process attributes across a spawn operation. IEEE Std 1003.1-200x does not define comparison or assignment operators for the type **posix_spawnattr_t**.

Each implementation shall document the individual attributes it uses and their default values unless these values are defined by IEEE Std 1003.1-200x. Attributes not defined by IEEE Std 1003.1-200x, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

The resulting spawn attributes object (possibly modified by setting individual attribute values), is used to modify the behavior of *posix_spawn*( ) or *posix_spawnp*( ). After a spawn attributes object has been used to spawn a process by a call to a *posix_spawn*( ) or *posix_spawnp*( ), any function affecting the attributes object (including destruction) shall not affect any process that has been spawned in this way.

RETURN VALUE

Upon successful completion, *posix_spawnattr_destroy*( ) and *posix_spawnattr_init*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *posix_spawnattr_init*( ) function shall fail if:

[ENOMEM] Insufficient memory exists to initialize the spawn attributes object.

The *posix_spawnattr_destroy*( ) function may fail if:

[EINVAL] The value specified by attr is invalid.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

The original spawn interface proposed in IEEE Std 1003.1-200x defined the attributes that specify the inheritance of process attributes across a spawn operation as a structure. In order to be able to separate optional individual attributes under their appropriate options (that is, the *spawn-schedparam* and *spawn-schedpolicy* attributes depending upon the Process Scheduling option), and

also for extensibility and consistency with the newer POSIX interfaces, the attributes interface has been changed to an opaque data type. This interface now consists of the type **posix_spawnattr_t**, representing a spawn attributes object, together with associated functions to initialize or destroy the attributes object, and to set or get each individual attribute. Although the new object-oriented interface is more verbose than the original structure, it is simple to use, more extensible, and easy to implement.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_spawn( ), *posix_spawnattr_getsigdefault*( ), *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setsigmask*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #106 is applied, noting that the effect of initializing an already initialized spawn attributes option is undefined. |

29029 **NAME**

29030 posix_spawnattr_getflags, posix_spawnattr_setflags — get and set spawn-flags attribute of
29031 spawn attributes object (**ADVANCED REALTIME**)

29032 **SYNOPSIS**

29033 SPN
```
#include <spawn.h>

int posix_spawnattr_getflags(const posix_spawnattr_t *restrict attr,
    short *restrict flags);
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
```
29037

29038 **DESCRIPTION**

29039 The *posix_spawnattr_getflags*( ) function shall obtain the value of the *spawn-flags* attribute from |
29040 the attributes object referenced by *attr*. |

29041 The *posix_spawnattr_setflags*( ) function shall set the *spawn-flags* attribute in an initialized |
29042 attributes object referenced by *attr*. |

29043 The *spawn-flags* attribute is used to indicate which process attributes are to be changed in the |
29044 new process image when invoking *posix_spawn*( ) or *posix_spawnp*( ). It is the bitwise-inclusive
29045 OR of zero or more of the following flags:

29046 POSIX_SPAWN_RESETIDS
29047 POSIX_SPAWN_SETPGROUP
29048 POSIX_SPAWN_SETSIGDEF
29049 POSIX_SPAWN_SETSIGMASK
29050 PS POSIX_SPAWN_SETSCHEDPARAM
29051 POSIX_SPAWN_SETSCHEDULER
29052

29053 These flags are defined in **<spawn.h>**. The default value of this attribute shall be as if no flags |
29054 were set. |

29055 **RETURN VALUE**

29056 Upon successful completion, *posix_spawnattr_getflags*( ) shall return zero and store the value of
29057 the *spawn-flags* attribute of *attr* into the object referenced by the *flags* parameter; otherwise, an
29058 error number shall be returned to indicate the error.

29059 Upon successful completion, *posix_spawnattr_setflags*( ) shall return zero; otherwise, an error
29060 number shall be returned to indicate the error.

29061 **ERRORS**

29062 These functions may fail if:

29063 [EINVAL] The value specified by *attr* is invalid.

29064 The *posix_spawnattr_setflags*( ) function may fail if:

29065 [EINVAL] The value of the attribute being set is not valid.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_spawn( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getsigdefault*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29082 **NAME**

29083 posix_spawnattr_getpgroup, posix_spawnattr_setpgroup — get and set spawn-pgroup attribute
29084 of spawn attributes object (**ADVANCED REALTIME**)

29085 **SYNOPSIS**

29086 SPN
```
#include <spawn.h>

int posix_spawnattr_getpgroup(const posix_spawnattr_t *restrict attr,
    pid_t *restrict pgroup);
int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```
29087
29088
29089
29090

29091 **DESCRIPTION**

29092 The *posix_spawnattr_getpgroup*( ) function shall obtain the value of the *spawn-pgroup* attribute |
29093 from the attributes object referenced by *attr*.

29094 The *posix_spawnattr_setpgroup*( ) function shall set the *spawn-pgroup* attribute in an initialized |
29095 attributes object referenced by *attr*. |

29096 The *spawn-pgroup* attribute represents the process group to be joined by the new process image |
29097 in a spawn operation (if POSIX_SPAWN_SETPGROUP is set in the *spawn-flags* attribute). The |
29098 default value of this attribute shall be zero. |

29099 **RETURN VALUE**

29100 Upon successful completion, *posix_spawnattr_getpgroup*( ) shall return zero and store the value of
29101 the *spawn-pgroup* attribute of *attr* into the object referenced by the *pgroup* parameter; otherwise,
29102 an error number shall be returned to indicate the error.

29103 Upon successful completion, *posix_spawnattr_setpgroup*( ) shall return zero; otherwise, an error
29104 number shall be returned to indicate the error.

29105 **ERRORS**

29106 These functions may fail if:

29107 [EINVAL] The value specified by *attr* is invalid.

29108 The *posix_spawnattr_setpgroup*( ) function may fail if:

29109 [EINVAL] The value of the attribute being set is not valid.

29110 **EXAMPLES**

29111 None.

29112 **APPLICATION USAGE**

29113 These functions are part of the Spawn option and need not be provided on all implementations.

29114 **RATIONALE**

29115 None.

29116 **FUTURE DIRECTIONS**

29117 None.

29118 **SEE ALSO**

29119 *posix_spawn*( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getsigdefault*( ),
29120 *posix_spawnattr_getflags*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ),
29121 *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ),
29122 *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ),
29123 *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

29124 **CHANGE HISTORY**
29125 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29126 **NAME**

29127 posix_spawnattr_getschedparam, posix_spawnattr_setschedparam — get and set spawn-
29128 schedparam attribute of spawn attributes object (**ADVANCED REALTIME**)

29129 **SYNOPSIS**

29130 SPN PS
```
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedparam(
    const posix_spawnattr_t *restrict attr,
    struct sched_param *restrict schedparam);
int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
    const struct sched_param *restrict schedparam);
```
29137

29138 **DESCRIPTION**

29139 The *posix_spawnattr_getschedparam*( ) function shall obtain the value of the *spawn-schedparam* |
29140 attribute from the attributes object referenced by *attr*.

29141 The *posix_spawnattr_setschedparam*( ) function shall set the *spawn-schedparam* attribute in an |
29142 initialized attributes object referenced by *attr*. |

29143 The *spawn-schedparam* attribute represents the scheduling parameters to be assigned to the new |
29144 process image in a spawn operation (if POSIX_SPAWN_SETSCHEDULER or |
29145 POSIX_SPAWN_SETSCHEDPARAM is set in the *spawn-flags* attribute). The default value of this |
29146 attribute is unspecified. |

29147 **RETURN VALUE**

29148 Upon successful completion, *posix_spawnattr_getschedparam*( ) shall return zero and store the
29149 value of the *spawn-schedparam* attribute of *attr* into the object referenced by the *schedparam*
29150 parameter; otherwise, an error number shall be returned to indicate the error.

29151 Upon successful completion, *posix_spawnattr_setschedparam*( ) shall return zero; otherwise, an
29152 error number shall be returned to indicate the error.

29153 **ERRORS**

29154 These functions may fail if:

29155 [EINVAL] The value specified by *attr* is invalid.

29156 The *posix_spawnattr_setschedparam*( ) function may fail if:

29157 [EINVAL] The value of the attribute being set is not valid.

29158 **EXAMPLES**

29159 None.

29160 **APPLICATION USAGE**

29161 These functions are part of the Spawn and Process Scheduling options and need not be provided
29162 on all implementations.

29163 **RATIONALE**

29164 None.

29165 **FUTURE DIRECTIONS**

29166 None.

SEE ALSO

posix_spawn( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getsigdefault*( ), *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29175 **NAME**

29176 posix_spawnattr_getschedpolicy, posix_spawnattr_setschedpolicy — get and set spawn-
29177 schedpolicy attribute of spawn attributes object (**ADVANCED REALTIME**)

29178 **SYNOPSIS**

29179 SPN PS
```
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedpolicy(
    const posix_spawnattr_t *restrict attr,
    int *restrict schedpolicy);
int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
    int schedpolicy);
```

29186

29187 **DESCRIPTION**

29188 The *posix_spawnattr_getschedpolicy*( ) function shall obtain the value of the *spawn-schedpolicy* |
29189 attribute from the attributes object referenced by *attr*.

29190 The *posix_spawnattr_setschedpolicy*( ) function shall set the *spawn-schedpolicy* attribute in an |
29191 initialized attributes object referenced by *attr*. |

29192 The *spawn-schedpolicy* attribute represents the scheduling policy to be assigned to the new |
29193 process image in a spawn operation (if POSIX_SPAWN_SETSCHEDULER is set in the *spawn-* |
29194 *flags* attribute). The default value of this attribute is unspecified. |

29195 **RETURN VALUE**

29196 Upon successful completion, *posix_spawnattr_getschedpolicy*( ) shall return zero and store the
29197 value of the *spawn-schedpolicy* attribute of *attr* into the object referenced by the *schedpolicy*
29198 parameter; otherwise, an error number shall be returned to indicate the error.

29199 Upon successful completion, *posix_spawnattr_setschedpolicy*( ) shall return zero; otherwise, an
29200 error number shall be returned to indicate the error.

29201 **ERRORS**

29202 These functions may fail if:

29203 [EINVAL] The value specified by *attr* is invalid.

29204 The *posix_spawnattr_setschedpolicy*( ) function may fail if:

29205 [EINVAL] The value of the attribute being set is not valid.

29206 **EXAMPLES**

29207 None.

29208 **APPLICATION USAGE**

29209 These functions are part of the Spawn and Process Scheduling options and need not be provided
29210 on all implementations.

29211 **RATIONALE**

29212 None.

29213 **FUTURE DIRECTIONS**

29214 None.

29215 **SEE ALSO**

29216 *posix_spawn*( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getsigdefault*( ),
29217 *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ),
29218 *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ),

29219 *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setsigmask*( ),
29220 *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**, **<spawn.h>**

29221 **CHANGE HISTORY**

29222 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29223 **NAME**

29224 posix_spawnattr_getsigdefault, posix_spawnattr_setsigdefault — get and set spawn-sigdefault
29225 attribute of spawn attributes object (**ADVANCED REALTIME**)

29226 **SYNOPSIS**

29227 SPN
```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigdefault(
    const posix_spawnattr_t *restrict attr,
    sigset_t *restrict sigdefault);
int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigdefault);
```
29234

29235 **DESCRIPTION**

29236 The *posix_spawnattr_getsigdefault*( ) function shall obtain the value of the *spawn-sigdefault* |
29237 attribute from the attributes object referenced by *attr*.

29238 The *posix_spawnattr_setsigdefault*( ) function shall set the *spawn-sigdefault* attribute in an |
29239 initialized attributes object referenced by *attr*. |

29240 The *spawn-sigdefault* attribute represents the set of signals to be forced to default signal handling |
29241 in the new process image (if POSIX_SPAWN_SETSIGDEF is set in the *spawn-flags* attribute) by a |
29242 spawn operation. The default value of this attribute shall be an empty signal set. |

29243 **RETURN VALUE**

29244 Upon successful completion, *posix_spawnattr_getsigdefault*( ) shall return zero and store the value
29245 of the *spawn-sigdefault* attribute of *attr* into the object referenced by the *sigdefault* parameter;
29246 otherwise, an error number shall be returned to indicate the error.

29247 Upon successful completion, *posix_spawnattr_setsigdefault*( ) shall return zero; otherwise, an error
29248 number shall be returned to indicate the error.

29249 **ERRORS**

29250 These functions may fail if:

29251 [EINVAL] The value specified by *attr* is invalid.

29252 The *posix_spawnattr_setsigdefault*( ) function may fail if:

29253 [EINVAL] The value of the attribute being set is not valid.

29254 **EXAMPLES**

29255 None.

29256 **APPLICATION USAGE**

29257 These functions are part of the Spawn option and need not be provided on all implementations.

29258 **RATIONALE**

29259 None.

29260 **FUTURE DIRECTIONS**

29261 None.

29262 **SEE ALSO**

29263 *posix_spawn*( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getflags*( ),
29264 *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ),
29265 *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ),
29266 *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ),

posix_spawnp( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29270 **NAME**

29271 posix_spawnattr_getsigmask, posix_spawnattr_setsigmask — get and set spawn-sigmask
29272 attribute of spawn attributes object (**ADVANCED REALTIME**)

29273 **SYNOPSIS**

29274 SPN
```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigmask(const posix_spawnattr_t *restrict attr,
    sigset_t *restrict sigmask);
int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigmask);
```
29280

29281 **DESCRIPTION**

29282 The *posix_spawnattr_getsigmask*( ) function shall obtain the value of the *spawn-sigmask* attribute |
29283 from the attributes object referenced by *attr*.

29284 The *posix_spawnattr_setsigmask*( ) function shall set the *spawn-sigmask* attribute in an initialized |
29285 attributes object referenced by *attr*. |

29286 The *spawn-sigmask* attribute represents the signal mask in effect in the new process image of a |
29287 spawn operation (if POSIX_SPAWN_SETSIGMASK is set in the *spawn-flags* attribute). The |
29288 default value of this attribute is unspecified. |

29289 **RETURN VALUE**

29290 Upon successful completion, *posix_spawnattr_getsigmask*( ) shall return zero and store the value
29291 of the *spawn-sigmask* attribute of *attr* into the object referenced by the *sigmask* parameter;
29292 otherwise, an error number shall be returned to indicate the error.

29293 Upon successful completion, *posix_spawnattr_setsigmask*( ) shall return zero; otherwise, an error
29294 number shall be returned to indicate the error.

29295 **ERRORS**

29296 These functions may fail if:

29297 [EINVAL] The value specified by *attr* is invalid.

29298 The *posix_spawnattr_setsigmask*( ) function may fail if:

29299 [EINVAL] The value of the attribute being set is not valid.

29300 **EXAMPLES**

29301 None.

29302 **APPLICATION USAGE**

29303 These functions are part of the Spawn option and need not be provided on all implementations.

29304 **RATIONALE**

29305 None.

29306 **FUTURE DIRECTIONS**

29307 None.

29308 **SEE ALSO**

29309 *posix_spawn*( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getsigdefault*( ),
29310 *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ),
29311 *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ),
29312 *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ),
29313 *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<spawn.h>**

29314 **CHANGE HISTORY**

29315 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29316 **NAME**

29317 posix_spawnattr_init — initialize spawn attributes object (**ADVANCED REALTIME**)

29318 **SYNOPSIS**

29319 SPN
```
#include <spawn.h>

int posix_spawnattr_init(posix_spawnattr_t *attr);
```
29320

29321

29322 **DESCRIPTION**

29323 Refer to *posix_spawnattr_destroy*( ).

29324 **NAME**

29325 posix_spawnattr_setflags — set spawn-flags attribute of spawn attributes object (**ADVANCED**
29326 **REALTIME**)

29327 **SYNOPSIS**

29328 SPN
```
#include <spawn.h>
```

29329
```
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
```

29330

29331 **DESCRIPTION**

29332 Refer to *posix_spawnattr_getflags*( ).

29333 **NAME**

29334 posix_spawnattr_setpgroup — set spawn-pgroup attribute of spawn attributes object
29335 (**ADVANCED REALTIME**)

29336 **SYNOPSIS**

29337 SPN
```
#include <spawn.h>
```

29338
```
int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

29339

29340 **DESCRIPTION**

29341 Refer to *posix_spawnattr_getpgroup*( ).

29342 **NAME**

29343 posix_spawnattr_setschedparam — set spawn-schedparam attribute of spawn attributes object
29344 (**ADVANCED REALTIME**)

29345 **SYNOPSIS**

29346 SPN PS
```
#include <sched.h>
#include <spawn.h>

int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
    const struct sched_param *restrict schedparam);
```
29347
29348
29349
29350

29351 **DESCRIPTION**

29352 Refer to *posix_spawnattr_getschedparam*( ).

29353 **NAME**

29354 posix_spawnattr_setschedpolicy — set spawn-schedpolicy attribute of spawn attributes object
29355 (**ADVANCED REALTIME**)

29356 **SYNOPSIS**

29357 SPN PS
```
#include <sched.h>
#include <spawn.h>

int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
    int schedpolicy);
```
29358
29359
29360
29361

29362 **DESCRIPTION**

29363 Refer to *posix_spawnattr_getschedpolicy*( ).

29364 **NAME**

29365 posix_spawnattr_setsigdefault — set spawn-sigdefault attribute of spawn attributes object
29366 (**ADVANCED REALTIME**)

29367 **SYNOPSIS**

29368 SPN
```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigdefault);
```
29369

29370

29371

29372

29373 **DESCRIPTION**

29374 Refer to *posix_spawnattr_getsigdefault*( ).

29375 **NAME**

29376 posix_spawnattr_setsigmask — set spawn-sigmask attribute of spawn attributes object
29377 (**ADVANCED REALTIME**)

29378 **SYNOPSIS**

29379 SPN
```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigmask);
```
29383

29384 **DESCRIPTION**

29385 Refer to *posix_spawnattr_getsigmask*( ).

29386 **NAME**

29387 posix_spawnp — spawn a process (**ADVANCED REALTIME**)

29388 **SYNOPSIS**

29389 SPN

```
#include <spawn.h>

int posix_spawnp(pid_t *restrict pid, const char *restrict file,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp,
    char *const argv[restrict], char *const envp[restrict]);
```

29390

29391

29392

29393

29394

29395 **DESCRIPTION**

29396 Refer to *posix_spawn*( ).

29397 **NAME**

29398 posix_trace_attr_destroy, posix_trace_attr_init — trace stream attributes object destroy and
29399 initialization (**TRACING**)

29400 **SYNOPSIS**

29401 TRC
```
#include <trace.h>

int posix_trace_attr_destroy(trace_attr_t *attr);
int posix_trace_attr_init(trace_attr_t *attr);
```
29402
29403
29404

29405 **DESCRIPTION**

29406 The *posix_trace_attr_destroy*( ) function shall destroy an initialized trace attributes object. A |
29407 destroyed *attr* attributes object can be reinitialized using *posix_trace_attr_init*( ); the results of |
29408 otherwise referencing the object after it has been destroyed are undefined. |

29409 The *posix_trace_attr_init*( ) function shall initialize a trace attributes object *attr* with the default
29410 value for all of the individual attributes used by a given implementation. The read-only
29411 *generation-version* and *clock-resolution* attributes of the newly initialized trace attributes object
29412 shall be set to their appropriate values (see Section 2.11.1.2 (on page 525)).

29413 Results are undefined if *posix_trace_attr_init*( ) is called specifying an already initialized *attr* |
29414 attributes object. |

29415 Implementations may add extensions to the trace attributes object structure as permitted in the
29416 Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance.

29417 The resulting attributes object (possibly modified by setting individual attributes values), when
29418 used by *posix_trace_create*( ), defines the attributes of the trace stream created. A single attributes
29419 object can be used in multiple calls to *posix_trace_create*( ). After one or more trace streams have
29420 been created using an attributes object, any function affecting that attributes object, including |
29421 destruction, shall not affect any trace stream previously created. An initialized attributes object |
29422 also serves to receive the attributes of an existing trace stream or trace log when calling the
29423 *posix_trace_get_attr*( ) function.

29424 **RETURN VALUE**

29425 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
29426 return the corresponding error number.

29427 **ERRORS**

29428 The *posix_trace_attr_destroy*( ) function may fail if:

29429 [EINVAL] The value of *attr* is invalid.

29430 The *posix_trace_attr_init*( ) function shall fail if:

29431 [ENOMEM] Insufficient memory exists to initialize the trace attributes object.

29432 **EXAMPLES**

29433 None.

29434 **APPLICATION USAGE**

29435 None.

29436 **RATIONALE**

29437 None.

29438 **FUTURE DIRECTIONS**

29439 None.

29440 **SEE ALSO**

29441 *posix_trace_create*( ), *posix_trace_get_attr*( ), *uname*( ), the Base Definitions volume of
29442 IEEE Std 1003.1-200x, **<trace.h>**

29443 **CHANGE HISTORY**

29444 First released in Issue 6. Derived from IEEE Std 1003.1q-2000. |

29445 IEEE PASC Interpretation 1003.1 #123 is applied. |

29446 **NAME**

29447 posix_trace_attr_getclockres, posix_trace_attr_getcreatetime, posix_trace_attr_getgenversion,
29448 posix_trace_attr_getname, posix_trace_attr_setname — retrieve and set information about a
29449 trace stream (**TRACING**)

29450 **SYNOPSIS**

29451 TRC
```
#include <time.h>
#include <trace.h>

int posix_trace_attr_getclockres(const trace_attr_t *attr,
    struct timespec *resolution);
int posix_trace_attr_getcreatetime(const trace_attr_t *attr,
    struct timespec *createtime);

#include <trace.h>

int posix_trace_attr_getgenversion(const trace_attr_t *attr,
    char *genversion);
int posix_trace_attr_getname(const trace_attr_t *attr,
    char *tracename);
int posix_trace_attr_setname(trace_attr_t *attr,
    const char *tracename);
```
29464

29465 **DESCRIPTION**

29466 The *posix_trace_attr_getclockres*( ) function shall copy the clock resolution of the clock used to
29467 generate timestamps from the *clock-resolution* attribute of the attributes object pointed to by the
29468 *attr* argument into the structure pointed to by the *resolution* argument.

29469 The *posix_trace_attr_getcreatetime*( ) function shall copy the trace stream creation time from the
29470 *creation-time* attribute of the attributes object pointed to by the *attr* argument into the structure
29471 pointed to by the *createtime* argument. The *creation-time* attribute shall represent the time of
29472 creation of the trace stream.

29473 The *posix_trace_attr_getgenversion*( ) function shall copy the string containing version information
29474 from the *generation-version* attribute of the attributes object pointed to by the *attr* argument into
29475 the string pointed to by the *genversion* argument. The *genversion* argument shall be the address of
29476 a character array which can store at least {TRACE_NAME_MAX} characters.

29477 The *posix_trace_attr_getname*( ) function shall copy the string containing the trace name from the
29478 *trace-name* attribute of the attributes object pointed to by the *attr* argument into the string
29479 pointed to by the *tracename* argument. The *tracename* argument shall be the address of a character
29480 array which can store at least {TRACE_NAME_MAX} characters.

29481 The *posix_trace_attr_setname*( ) function shall set the name in the *trace-name* attribute of the
29482 attributes object pointed to by the *attr* argument, using the trace name string supplied by the
29483 *tracename* argument. If the supplied string contains more than {TRACE_NAME_MAX}
29484 characters, the name copied into the *trace-name* attribute may be truncated to one less than the
29485 length of {TRACE_NAME_MAX} characters. The default value is a null string.

29486 **RETURN VALUE**

29487 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
29488 return the corresponding error number.

29489 If successful, the *posix_trace_attr_getclockres*( ) function stores the *clock-resolution* attribute value
29490 in the object pointed to by *resolution*. Otherwise, the content of this object is unspecified.

If successful, the *posix_trace_attr_getcreatetime*( ) function stores the trace stream creation time in the object pointed to by *createtime*. Otherwise, the content of this object is unspecified.

If successful, the *posix_trace_attr_getgenversion*( ) function stores the trace version information in the string pointed to by *genversion*. Otherwise, the content of this string is unspecified.

If successful, the *posix_trace_attr_getname*( ) function stores the trace name in the string pointed to by *tracename*. Otherwise, the content of this string is unspecified.

ERRORS

The *posix_trace_attr_getclockres*( ), *posix_trace_attr_getcreatetime*( ), *posix_trace_attr_getgenversion*( ), and *posix_trace_attr_getname*( ) functions may fail if:

[EINVAL] The value specified by one of the arguments is invalid.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_attr_init( ), *posix_trace_create*( ), *posix_trace_get_attr*( ), *uname*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

29514 **NAME**

29515 posix_trace_attr_getcreatetime — retrieve and set information about a trace stream (**TRACING**)

29516 **SYNOPSIS**

29517 TRC
```
#include <time.h>
#include <trace.h>

int posix_trace_attr_getcreatetime(const trace_attr_t *attr,
    struct timespec *createtime);
```
29521

29522 **DESCRIPTION**

29523 Refer to *posix_trace_attr_getclockres*( ).

29524 **NAME**

29525 posix_trace_attr_getgenversion — retrieve and set information about a trace stream
29526 (**TRACING**)

29527 **SYNOPSIS**

29528 TRC
```
#include <trace.h>
```

29529
```
int posix_trace_attr_getgenversion(const trace_attr_t *attr,
    char *genversion);
```

29530
29531

29532 **DESCRIPTION**

29533 Refer to *posix_trace_attr_getclockres*( ).

NAME

posix_trace_attr_getinherited, posix_trace_attr_getlogfullpolicy, posix_trace_attr_getstreamfullpolicy, posix_trace_attr_setinherited, posix_trace_attr_setlogfullpolicy, posix_trace_attr_setstreamfullpolicy — retrieve and set the behavior of a trace stream (**TRACING**)

SYNOPSIS

TRC
```
#include <trace.h>
```

TRC TRI
```
int posix_trace_attr_getinherited(const trace_attr_t *restrict attr,
    int *restrict inheritancepolicy);
```
TRC TRL
```
int posix_trace_attr_getlogfullpolicy(const trace_attr_t *restrict attr,
    int *restrict logpolicy);
```
TRC
```
int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *attr,
    int *streampolicy);
```
TRC TRI
```
int posix_trace_attr_setinherited(trace_attr_t *attr,
    int inheritancepolicy);
```
TRC TRL
```
int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
    int logpolicy);
```
TRC
```
int posix_trace_attr_setstreamfullpolicy(trace_attr_t *attr,
    int streampolicy);
```

DESCRIPTION

TRI The *posix_trace_attr_getinherited*( ) and *posix_trace_attr_setinherited*( ) functions, respectively, shall get and set the inheritance policy stored in the *inheritance* attribute for traced processes across the *fork*( ) and *spawn*( ) operations. The *inheritance* attribute of the attributes object pointed to by the *attr* argument shall be set to one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_CLOSE_FOR_CHILD
: After a *fork*( ) or *spawn*( ) operation, the child shall not be traced, and tracing of the parent shall continue.

POSIX_TRACE_INHERITED
: After a *fork*( ) or *spawn*( ) operation, if the parent is being traced, its child shall be concurrently traced using the same trace stream.

The default value for the *inheritance* attribute is POSIX_TRACE_CLOSE_FOR_CHILD.

TRL The *posix_trace_attr_getlogfullpolicy*( ) and *posix_trace_attr_setlogfullpolicy*( ) functions, respectively, shall get and set the trace log full policy stored in the *log-full-policy* attribute of the attributes object pointed to by the *attr* argument.

The *log-full-policy* attribute shall be set to one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_LOOP
: The trace log shall loop until the associated trace stream is stopped. This policy means that when the trace log gets full, the file system shall reuse the resources allocated to the oldest trace events that were recorded. In this way, the trace log will always contain the most recent trace events flushed.

POSIX_TRACE_UNTIL_FULL
: The trace stream shall be flushed to the trace log until the trace log is full. This condition can be deduced from the *posix_log_full_status* member status (see the *posix_trace_status_info*( ) function). The last recorded trace event shall be the POSIX_TRACE_STOP trace event.

POSIX_TRACE_APPEND
The associated trace stream shall be flushed to the trace log without log size limitation. If the application specifies POSIX_TRACE_APPEND, the implementation shall ignore the *log-max-size* attribute.

The default value for the *log-full-policy* attribute is POSIX_TRACE_LOOP.

The *posix_trace_attr_getstreamfullpolicy*( ) and *posix_trace_attr_setstreamfullpolicy*( ) functions, respectively, shall get and set the trace stream full policy stored in the *stream-full-policy* attribute of the attributes object pointed to by the *attr* argument.

The *stream-full-policy* attribute shall be set to one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_LOOP
The trace stream shall loop until explicitly stopped by the *posix_trace_stop*( ) function. This policy means that when the trace stream is full, the trace system shall reuse the resources allocated to the oldest trace events recorded. In this way, the trace stream will always contain the most recent trace events recorded.

POSIX_TRACE_UNTIL_FULL
The trace stream will run until the trace stream resources are exhausted. Then the trace stream will stop. This condition can be deduced from *posix_stream_status* and *posix_stream_full_status* statuses (see the *posix_trace_status_info*( ) function). When this trace stream is read, a POSIX_TRACE_STOP trace event shall be reported after reporting the last recorded trace event. The trace system shall reuse the resources allocated to any trace events already reported—see the *posix_trace_getnext_event*( ), *posix_trace_trygetnext_event*( ), and *posix_trace_timedgetnext_event*( ) functions—or already flushed for an active trace stream with log if the Trace Log option is supported; see the *posix_trace_flush*( ) function. The trace system shall restart the trace stream when it is empty and may restart it sooner. A POSIX_TRACE_START trace event shall be reported before reporting the next recorded trace event.

TRL POSIX_TRACE_FLUSH
If the Trace Log option is supported, this policy is identical to the POSIX_TRACE_UNTIL_FULL trace stream full policy except that the trace stream shall be flushed regularly as if *posix_trace_flush*( ) had been explicitly called. Defining this policy for an active trace stream without log shall be invalid.

The default value for the *stream-full-policy* attribute shall be POSIX_TRACE_LOOP for an active trace stream without log.

TRL If the Trace Log option is supported, the default value for the *stream-full-policy* attribute shall be POSIX_TRACE_FLUSH for an active trace stream with log.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

TRI If successful, the *posix_trace_attr_getinherited*( ) function shall store the *inheritance* attribute value in the object pointed to by *inheritancepolicy*. Otherwise, the content of this object is undefined.

TRL If successful, the *posix_trace_attr_getlogfullpolicy*( ) function shall store the *log-full-policy* attribute value in the object pointed to by *logpolicy*. Otherwise, the content of this object is undefined.

If successful, the *posix_trace_attr_getstreamfullpolicy*( ) function shall store the *stream-full-policy* attribute value in the object pointed to by *streampolicy*. Otherwise, the content of this object is undefined.

29627 **ERRORS**

29628 These functions may fail if:

29629 [EINVAL] The value specified by at least one of the arguments is invalid.

29630 **EXAMPLES**

29631 None.

29632 **APPLICATION USAGE**

29633 None.

29634 **RATIONALE**

29635 None.

29636 **FUTURE DIRECTIONS**

29637 None.

29638 **SEE ALSO**

29639 *fork*( ), *posix_trace_attr_init*( ), *posix_trace_create*( ), *posix_trace_flush*( ), *posix_trace_get_attr*( ),
29640 *posix_trace_getnext_event*( ), *posix_trace_start*( ), **posix_trace_status_info Structure**,
29641 *posix_trace_timedgetnext_event*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

29642 **CHANGE HISTORY**

29643 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

29644 **NAME**

29645 posix_trace_attr_getlogfullpolicy — retrieve and set the behavior of a trace stream (**TRACING**)

29646 **SYNOPSIS**

29647 TRC
```
#include <trace.h>
```

29648 TRC TRL
```
int posix_trace_attr_getlogfullpolicy(const trace_attr_t *restrict attr,
    int *restrict logpolicy);
```

29650

29651 **DESCRIPTION**

29652 Refer to *posix_trace_attr_getinherited*( ).

NAME

posix_trace_attr_getlogsize, posix_trace_attr_getmaxdatasize, posix_trace_attr_getmaxsystemeventsize, posix_trace_attr_getmaxuserevent size, posix_trace_attr_getstreamsize, posix_trace_attr_setlogsize, posix_trace_attr_setmaxdatasize, posix_trace_attr_setstreamsize — retrieve and set trace stream size attributes (**TRACING**)

SYNOPSIS

TRC
```
#include <sys/types.h>
#include <trace.h>
```

TRC TRL
```
int posix_trace_attr_getlogsize(const trace_attr_t *restrict attr,
    size_t *restrict logsize);
```
TRC
```
int posix_trace_attr_getmaxdatasize(const trace_attr_t *restrict attr,
    size_t *restrict maxdatasize);
int posix_trace_attr_getmaxsystemeventsize(
    const trace_attr_t *restrict attr,
    size_t *restrict eventsize);
int posix_trace_attr_getmaxuserevent size(
    const trace_attr_t *restrict attr,
    size_t data_len, size_t *restrict eventsize);
int posix_trace_attr_getstreamsize(const trace_attr_t *restrict attr,
    size_t *restrict streamsize);
```
TRC TRL
```
int posix_trace_attr_setlogsize(trace_attr_t *attr,
    size_t logsize);
```
TRC
```
int posix_trace_attr_setmaxdatasize(trace_attr_t *attr,
    size_t maxdatasize);
int posix_trace_attr_setstreamsize(trace_attr_t *attr,
    size_t streamsize);
```

DESCRIPTION

TRL
The *posix_trace_attr_getlogsize*( ) function shall copy the log size, in bytes, from the *log-max-size* attribute of the attributes object pointed to by the *attr* argument into the variable pointed to by the *logsize* argument. This log size is the maximum total of bytes that shall be allocated for system and user trace events in the trace log. The default value for the *log-max-size* attribute is implementation-defined.

The *posix_trace_attr_setlogsize*( ) function shall set the maximum allowed size, in bytes, in the *log-max-size* attribute of the attributes object pointed to by the *attr* argument, using the size value supplied by the *logsize* argument.

The trace log size shall be used if the *log-full-policy* attribute is set to POSIX_TRACE_LOOP or POSIX_TRACE_UNTIL_FULL. If the *log-full-policy* attribute is set to POSIX_TRACE_APPEND, the implementation shall ignore the *log-max-size* attribute.

The *posix_trace_attr_getmaxdatasize*( ) function shall copy the maximum user trace event data size, in bytes, from the *max-data-size* attribute of the attributes object pointed to by the *attr* argument into the variable pointed to by the *maxdatasize* argument. The default value for the *max-data-size* attribute is implementation-defined.

The *posix_trace_attr_getmaxsystemeventsize*( ) function shall calculate the maximum memory size, in bytes, required to store a single system trace event. This value is calculated for the trace stream attributes object pointed to by the *attr* argument and is returned in the variable pointed to by the *eventsize* argument.

The values returned as the maximum memory sizes of the user and system trace events shall be such that if the sum of the maximum memory sizes of a set of the trace events that may be recorded in a trace stream is less than or equal to the *stream-min-size* attribute of that trace stream, the system provides the necessary resources for recording all those trace events, without loss.

The *posix_trace_attr_getmaxusereventsize*( ) function shall calculate the maximum memory size, in bytes, required to store a single user trace event generated by a call to *posix_trace_event*( ) with a *data_len* parameter equal to the *data_len* value specified in this call. This value is calculated for the trace stream attributes object pointed to by the *attr* argument and is returned in the variable pointed to by the *eventsize* argument.

The *posix_trace_attr_getstreamsize*( ) function shall copy the stream size, in bytes, from the *stream-min-size* attribute of the attributes object pointed to by the *attr* argument into the variable pointed to by the *streamsize* argument.

This stream size is the current total memory size reserved for system and user trace events in the trace stream. The default value for the *stream-min-size* attribute is implementation-defined. The stream size refers to memory used to store trace event records. Other stream data (for example, trace attribute values) shall not be included in this size.

The *posix_trace_attr_setmaxdatasize*( ) function shall set the maximum allowed size, in bytes, in the *max-data-size* attribute of the attributes object pointed to by the *attr* argument, using the size value supplied by the *maxdatasize* argument. This maximum size is the maximum allowed size for the user data argument which may be passed to *posix_trace_event*( ). The implementation shall be allowed to truncate data passed to *trace_user_event* which is longer than *maxdatasize*.

The *posix_trace_attr_setstreamsize*( ) function shall set the minimum allowed size, in bytes, in the *stream-min-size* attribute of the attributes object pointed to by the *attr* argument, using the size value supplied by the *streamsize* argument.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

TRL The *posix_trace_attr_getlogsize*( ) function stores the maximum trace log allowed size in the object pointed to by *logsize*, if successful.

The *posix_trace_attr_getmaxdatasize*( ) function stores the maximum trace event record memory size in the object pointed to by *maxdatasize*, if successful.

The *posix_trace_attr_getmaxsystemeventsize*( ) function stores the maximum memory size to store a single system trace event in the object pointed to by *eventsize*, if successful.

The *posix_trace_attr_getmaxusereventsize*( ) function stores the maximum memory size to store a single user trace event in the object pointed to by *eventsize*, if successful.

The *posix_trace_attr_getstreamsize*( ) function stores the maximum trace stream allowed size in the object pointed to by *streamsize*, if successful.

ERRORS

These functions may fail if:

[EINVAL] The value specified by one of the arguments is invalid.

29741 **EXAMPLES**

29742 None.

29743 **APPLICATION USAGE**

29744 None.

29745 **RATIONALE**

29746 None.

29747 **FUTURE DIRECTIONS**

29748 None.

29749 **SEE ALSO**

29750 *posix_trace_attr_init*( ), *posix_trace_create*( ), *posix_trace_event*( ), *posix_trace_get_attr*( ), the Base
29751 Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

29752 **CHANGE HISTORY**

29753 First released in Issue 6. Derived from the IEEE Std 1003.1q-2000.

29754 **NAME**

29755 posix_trace_attr_getmaxdatasize, posix_trace_attr_getmaxsystemeventsize,
29756 posix_trace_attr_getmaxuserevent size — retrieve and set trace stream size attributes
29757 (**TRACING**)

29758 **SYNOPSIS**

29759 TRC
```
#include <sys/types.h>
#include <trace.h>
```

29761 TRC
```
int posix_trace_attr_getmaxdatasize(const trace_attr_t *restrict attr,
    size_t *restrict maxdatasize);
int posix_trace_attr_getmaxsystemeventsize(
    const trace_attr_t *restrict attr,
    size_t *restrict eventsize);
int posix_trace_attr_getmaxuserevent size(
    const trace_attr_t *restrict attr,
    size_t data_len, size_t *restrict eventsize);
```

29769

29770 **DESCRIPTION**

29771 Refer to *posix_trace_attr_getlogsize*( ).

29772 **NAME**

29773 posix_trace_attr_getname — retrieve and set information about a trace stream (**TRACING**)

29774 **SYNOPSIS**

29775 TRC
```
#include <trace.h>

int posix_trace_attr_getname(const trace_attr_t *attr,
    char *tracename);
```
29778

29779 **DESCRIPTION**

29780 Refer to *posix_trace_attr_getclockres*( ).

29781 **NAME**

29782 posix_trace_attr_getstreamfullpolicy — retrieve and set the behavior of a trace stream
29783 (**TRACING**)

29784 **SYNOPSIS**

29785 TRC
```
#include <trace.h>
```

29786
```
int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *attr,
    int *streampolicy);
```

29787

29788

29789 **DESCRIPTION**

29790 Refer to *posix_trace_attr_getinherited*( ).

29791 **NAME**

29792 posix_trace_attr_getstreamsize — retrieve and set trace stream size attributes (**TRACING**)

29793 **SYNOPSIS**

29794 TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_getstreamsize(const trace_attr_t *restrict attr,
    size_t *restrict streamsize);
```
29798

29799 **DESCRIPTION**

29800 Refer to *posix_trace_attr_getlogsize*( ).

29801 **NAME**

29802 posix_trace_attr_init — trace stream attributes object initialization (**TRACING**)

29803 **SYNOPSIS**

29804 TRC
```
#include <trace.h>
```

29805
```
int posix_trace_attr_init(trace_attr_t *attr);
```

29806

29807 **DESCRIPTION**

29808 Refer to *posix_trace_attr_destroy*( ).

29809 **NAME**

29810 posix_trace_attr_setinherited — retrieve and set the behavior of a trace stream (**TRACING**)

29811 **SYNOPSIS**

29812 TRC
```
#include <trace.h>
```

29813 TRC TRI
```
int posix_trace_attr_setinherited(trace_attr_t *attr,
    int inheritancepolicy);
```

29814

29815

29816 **DESCRIPTION**

29817 Refer to *posix_trace_attr_getinherited*( ).

29818 **NAME**

29819 posix_trace_attr_setlogfullpolicy — retrieve and set the behavior of a trace stream (**TRACING**)

29820 **SYNOPSIS**

29821 TRC
```
#include <trace.h>
```

29822 TRC TRL
```
int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
    int logpolicy);
```

29824

29825 **DESCRIPTION**

29826 Refer to *posix_trace_attr_getinherited*( ).

29827 **NAME**

29828 posix_trace_attr_setlogsize — retrieve and set trace stream size attributes (**TRACING**)

29829 **SYNOPSIS**

29830 TRC
```
#include <sys/types.h>
#include <trace.h>
```

29832 TRC TRL
```
int posix_trace_attr_setlogsize(trace_attr_t *attr,
    size_t logsize);
```

29834

29835 **DESCRIPTION**

29836 Refer to *posix_trace_attr_getlogsize*( ).

29837 **NAME**

29838 posix_trace_attr_setmaxdatasize — retrieve and set trace stream size attributes (**TRACING**)

29839 **SYNOPSIS**

29840 TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_setmaxdatasize(trace_attr_t *attr,
    size_t maxdatasize);
```
29844

29845 **DESCRIPTION**

29846 Refer to *posix_trace_attr_getlogsize*( ).

29847 **NAME**

29848 posix_trace_attr_setname — retrieve and set information about a trace stream (**TRACING**)

29849 **SYNOPSIS**

29850 TRC
```
#include <trace.h>
```

29851
```
int posix_trace_attr_setname(trace_attr_t *attr,
    const char *tracename);
```

29853

29854 **DESCRIPTION**

29855 Refer to *posix_trace_attr_getclockres*( ).

29856 **NAME**

29857 posix_trace_attr_setstreamfullpolicy — retrieve and set the behavior of a trace stream
29858 (**TRACING**)

29859 **SYNOPSIS**

29860 TRC
```
#include <trace.h>
```

29861 TRC TRL
```
int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
    int logpolicy);
```
29862
29863

29864 **DESCRIPTION**

29865 Refer to *posix_trace_attr_getinherited*( ).

29866 **NAME**

29867 posix_trace_attr_setstreamsize — retrieve and set trace stream size attributes (**TRACING**)

29868 **SYNOPSIS**

29869 TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_setstreamsize(trace_attr_t *attr,
    size_t streamsize);
```
29873

29874 **DESCRIPTION**

29875 Refer to *posix_trace_attr_getlogsize*( ).

29876 **NAME**

29877 posix_trace_clear — clear trace stream and trace log (**TRACING**)

29878 **SYNOPSIS**

29879 TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_clear(trace_id_t trid);
```
29880

29881

29882

29883 **DESCRIPTION**

29884 The *posix_trace_clear*( ) function shall reinitialize the trace stream identified by the argument *trid*
29885 as if it were returning from the *posix_trace_create*( ) function, except that the same allocated
29886 resources shall be reused, the mapping of trace event type identifiers to trace event names shall
29887 be unchanged, and the trace stream status shall remain unchanged (that is, if it was running, it
29888 remains running and if it was suspended, it remains suspended).

29889 All trace events in the trace stream recorded before the call to *posix_trace_clear*( ) shall be lost.
29890 The *posix_stream_full_status* status shall be set to POSIX_TRACE_NOT_FULL. There is no
29891 guarantee that all trace events that occurred during the *posix_trace_clear*( ) call are recorded; the
29892 behavior with respect to trace points that may occur during this call, is unspecified.

29893 TRL If the Trace Log option is supported and the trace stream has been created with a log, the
29894 *posix_trace_clear*( ) function shall reinitialize the trace stream with the same behavior as if the
29895 trace stream was created without the log, plus it shall reinitialize the trace log associated with
29896 the trace stream identified by the argument *trid* as if it were returning from the
29897 *posix_trace_create_withlog*( ) function, except that the same allocated resources, for the trace log,
29898 may be reused and the associated trace stream status remains unchanged. The first trace event
29899 recorded in the trace log after the call to *posix_trace_clear*( ) shall be the same as the first trace
29900 event recorded in the active trace stream after the call to *posix_trace_clear*( ). The
29901 *posix_log_full_status* status shall be set to POSIX_TRACE_NOT_FULL. There is no guarantee that
29902 all trace events that occurred during the *posix_trace_clear*( ) call are recorded in the trace log; the
29903 behavior with respect to trace points that may occur during this call is unspecified. If the log full
29904 policy is POSIX_TRACE_APPEND, the effect of a call to this function is unspecified for the trace
29905 log associated with the trace stream identified by the *trid* argument.

29906 **RETURN VALUE**

29907 Upon successful completion, the *posix_trace_clear*( ) function shall return a value of zero.
29908 Otherwise, it shall return the corresponding error number.

29909 **ERRORS**

29910 The *posix_trace_clear*( ) function shall fail if:

29911 [EINVAL] The value of the *trid* argument does not correspond to an active trace stream.

29912 **EXAMPLES**

29913 None.

29914 **APPLICATION USAGE**

29915 None.

29916 **RATIONALE**

29917 None.

29918 **FUTURE DIRECTIONS**

29919 None.

29920 **SEE ALSO**

29921 *posix_trace_attr_init*( ), *posix_trace_create*( ), *posix_trace_flush*( ), *posix_trace_get_attr*( ), the Base
29922 Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

29923 **CHANGE HISTORY**

29924 First released in Issue 6. Derived from IEEE Std 1003.1q-2000. |

29925 IEEE PASC Interpretation 1003.1 #123 is applied. |

29926 **NAME**

29927 posix_trace_close, posix_trace_open, posix_trace_rewind — trace log management (**TRACING**)

29928 **SYNOPSIS**

29929 TRC TRL
```
#include <trace.h>

int posix_trace_close(trace_id_t trid);
int posix_trace_open(int file_desc, trace_id_t *trid);
int posix_trace_rewind(trace_id_t trid);
```
29933

29934 **DESCRIPTION**

29935 The *posix_trace_close*( ) function shall deallocate the trace log identifier indicated by *trid*, and all
29936 of its associated resources. If there is no valid trace log pointed to by the *trid*, this function shall
29937 fail.

29938 The *posix_trace_open*( ) function shall allocate the necessary resources and establishes the
29939 connection between a trace log identified by the *file_desc* argument and a trace stream identifier
29940 identified by the object pointed to by the *trid* argument. The *file_desc* argument should be a valid
29941 open file descriptor that corresponds to a trace log. The *file_desc* argument shall be open for
29942 reading. The current trace event timestamp, which specifies the timestamp of the trace event
29943 that will be read by the next call to *posix_trace_getnext_event*( ), shall be set to the timestamp of
29944 the oldest trace event recorded in the trace log identified by *trid*.

29945 The *posix_trace_open*( ) function shall return a trace stream identifier in the variable pointed to by
29946 the *trid* argument, that may only be used by the following functions:

29947 *posix_trace_close*( ) *posix_trace_get_attr*( )
29948 *posix_trace_eventid_equal*( ) *posix_trace_get_status*( )
29949 *posix_trace_eventid_get_name*( ) *posix_trace_getnext_event*( )
29950 *posix_trace_eventtypelist_getnext_id*( ) *posix_trace_rewind*( )
29951 *posix_trace_eventtypelist_rewind*( )

29952 In particular, notice that the operations normally used by a trace controller process, such as
29953 *posix_trace_start*( ), *posix_trace_stop*( ), or *posix_trace_shutdown*( ), cannot be invoked using the
29954 trace stream identifier returned by the *posix_trace_open*( ) function.

29955 The *posix_trace_rewind*( ) function shall reset the current trace event timestamp, which specifies
29956 the timestamp of the trace event that will be read by the next call to *posix_trace_getnext_event*( ),
29957 to the timestamp of the oldest trace event recorded in the trace log identified by *trid*.

29958 **RETURN VALUE**

29959 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
29960 return the corresponding error number.

29961 If successful, the *posix_trace_open*( ) function stores the trace stream identifier value in the object
29962 pointed to by *trid*.

29963 **ERRORS**

29964 The *posix_trace_open*( ) function shall fail if: |

29965 [EINTR] The operation was interrupted by a signal and thus no trace log was opened.

29966 [EINVAL] The object pointed to by *file_desc* does not correspond to a valid trace log.

29967 The *posix_trace_close*( ) and *posix_trace_rewind*( ) functions may fail if:

29968 [EINVAL] The object pointed to by *trid* does not correspond to a valid trace log.

29969 **EXAMPLES**

29970 None.

29971 **APPLICATION USAGE**

29972 None.

29973 **RATIONALE**

29974 None.

29975 **FUTURE DIRECTIONS**

29976 None.

29977 **SEE ALSO**

29978 *posix_trace_get_attr*( ), *posix_trace_get_filter*( ), *posix_trace_getnext_event*( ), the Base Definitions
29979 volume of IEEE Std 1003.1-200x, **<trace.h>**

29980 **CHANGE HISTORY**

29981 First released in Issue 6. Derived from IEEE Std 1003.1q-2000. |

29982 IEEE PASC Interpretation 1003.1 #123 is applied. |

29983 **NAME**

29984 posix_trace_create, posix_trace_create_withlog, posix_trace_flush, posix_trace_shutdown —
29985 trace stream initialization, flush, and shutdown from a process (**TRACING**)

29986 **SYNOPSIS**

29987 TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_create(pid_t pid,
    const trace_attr_t *restrict attr,
    trace_id_t *restrict trid);
```
29992 TRC TRL
```
int posix_trace_create_withlog(pid_t pid,
    const trace_attr_t *restrict attr, int file_desc,
    trace_id_t *restrict trid);
int posix_trace_flush(trace_id_t trid);
```
29996 TRC
```
int posix_trace_shutdown(trace_id_t trid);
```
29997

29998 **DESCRIPTION**

29999 The *posix_trace_create*( ) function shall create an active trace stream. It allocates all the resources
30000 needed by the trace stream being created for tracing the process specified by *pid* in accordance
30001 with the *attr* argument. The *attr* argument represents the initial attributes of the trace stream and
30002 shall have been initialized by the function *posix_trace_attr_init*( ) prior the *posix_trace_create*( )
30003 call. If the argument *attr* is NULL, the default attributes shall be used. The *attr* attributes object
30004 shall be manipulated through a set of functions described in the *posix_trace_attr* family of
30005 functions. If the attributes of the object pointed to by *attr* are modified later, the attributes of the
30006 trace stream shall not be affected. The *creation-time* attribute of the newly created trace stream
30007 shall be set to the value of the system clock, if the Timers option is not supported, or to the value
30008 of the CLOCK_REALTIME clock, if the Timers option is supported.

30009 The *pid* argument represents the target process to be traced. If the process executing this
30010 function does not have appropriate privileges to trace the process identified by *pid*, an error shall
30011 be returned. If the *pid* argument is zero, the calling process shall be traced.

30012 The *posix_trace_create*( ) function shall store the trace stream identifier of the new trace stream in
30013 the object pointed to by the *trid* argument. This trace stream identifier shall be used in
30014 subsequent calls to control tracing. The *trid* argument may only be used by the following
30015 functions:

30016 *posix_trace_clear*( )	*posix_trace_getnext_event*( )
30017 *posix_trace_eventid_equal*( )	*posix_trace_shutdown*( )
30018 *posix_trace_eventid_get_name*( )	*posix_trace_start*( )
30019 *posix_trace_eventtypelist_getnext_id*( )	*posix_trace_stop*( )
30020 *posix_trace_eventtypelist_rewind*( )	*posix_trace_timedgetnext_event*( )
30021 *posix_trace_get_attr*( )	*posix_trace_trid_eventid_open*( )
30022 *posix_trace_get_status*( )	*posix_trace_trygetnext_event*( )

30023 TEF If the Trace Event Filter option is supported, the following additional functions may use the *trid*
30024 argument:

30025 *posix_trace_get_filter*( ) *posix_trace_set_filter*( )

30026

In particular, notice that the operations normally used by a trace analyzer process, such as *posix_trace_rewind*( ) or *posix_trace_close*( ), cannot be invoked using the trace stream identifier returned by the *posix_trace_create*( ) function.

TEF A trace stream shall be created in a suspended state. If the Trace Event Filter option is supported, its trace event type filter shall be empty.

The *posix_trace_create*( ) function may be called multiple times from the same or different processes, with the system-wide limit indicated by the runtime invariant value {TRACE_SYS_MAX}, which has the minimum value {_POSIX_TRACE_SYS_MAX}.

The trace stream identifier returned by the *posix_trace_create*( ) function in the argument pointed to by *trid* is valid only in the process that made the function call. If it is used from another process, that is a child process, in functions defined in IEEE Std 1003.1-200x, these functions shall return with the error [EINVAL].

TRL The *posix_trace_create_withlog*( ) function shall be equivalent to *posix_trace_create*( ), except that it associates a trace log with this stream. The *file_desc* argument shall be the file descriptor designating the trace log destination. The function shall fail if this file descriptor refers to a file with a file type that is not compatible with the log policy associated with the trace log. The list of the appropriate file types that are compatible with each log policy is implementation-defined.

The *posix_trace_create_withlog*( ) function shall return in the parameter pointed to by *trid* the trace stream identifier, which uniquely identifies the newly created trace stream, and shall be used in subsequent calls to control tracing. The *trid* argument may only be used by the following functions:

posix_trace_clear( )	*posix_trace_getnext_event*( )
posix_trace_eventid_equal( )	*posix_trace_shutdown*( )
posix_trace_eventid_get_name( )	*posix_trace_start*( )
posix_trace_eventtypelist_getnext_id( )	*posix_trace_stop*( )
posix_trace_eventtypelist_rewind( )	*posix_trace_timedgetnext_event*( )
posix_trace_flush( )	*posix_trace_trid_eventid_open*( )
posix_trace_get_attr( )	*posix_trace_trygetnext_event*( )
posix_trace_get_status( )	

TRL TEF If the Trace Event Filter option is supported, the following additional functions may use the *trid* argument:

posix_trace_get_filter( ) *posix_trace_set_filter*( )

TRL In particular, notice that the operations normally used by a trace analyzer process, such as *posix_trace_rewind*( ) or *posix_trace_close*( ), cannot be invoked using the trace stream identifier returned by the *posix_trace_create_withlog*( ) function.

The *posix_trace_flush*( ) function shall initiate a flush operation which copies the contents of the trace stream identified by the argument *trid* into the trace log associated with the trace stream at the creation time. If no trace log has been associated with the trace stream pointed to by *trid*, this function shall return an error. The termination of the flush operation can be polled by the *posix_trace_get_status*( ) function. During the flush operation, it shall be possible to trace new trace events up to the point when the trace stream becomes full. After flushing is completed, the space used by the flushed trace events shall be available for tracing new trace events.

If flushing the trace stream causes the resulting trace log to become full, the trace log full policy shall be applied. If the trace *log-full-policy* attribute is set, the following occurs:

POSIX_TRACE_UNTIL_FULL
: The trace events that have not yet been flushed shall be discarded.

POSIX_TRACE_LOOP
: The trace events that have not yet been flushed shall be written to the beginning of the trace log, overwriting previous trace events stored there.

POSIX_TRACE_APPEND
: The trace events that had not yet been flushed shall be appended to the trace log.

The *posix_trace_shutdown*( ) function shall stop the tracing of trace events in the trace stream identified by *trid*, as if *posix_trace_stop*( ) had been invoked. The *posix_trace_shutdown*( ) function shall free all the resources associated with the trace stream.

The *posix_trace_shutdown*( ) function shall not return until all the resources associated with the trace stream have been freed. When the *posix_trace_shutdown*( ) function returns, the *trid* argument becomes an invalid trace stream identifier. A call to this function shall unconditionally deallocate the resources regardless of whether all trace events have been retrieved by the analyzer process. Any thread blocked on one of the *trace_getnext_event*( ) functions (which specified this *trid*) before this call is unblocked with the error [EINVAL].

If the process exits, invokes an *exec*( ) call, or is terminated, the trace streams that the process had created and that have not yet been shut down, shall be automatically shut down as if an explicit call were made to the *posix_trace_shutdown*( ) function.

TRL For an active trace stream with log, when the *posix_trace_shutdown*( ) function is called, all trace events that have not yet been flushed to the trace log shall be flushed, as in the *posix_trace_flush*( ) function, and the trace log shall be closed.

When a trace log is closed, all the information that may be retrieved later from the trace log through the trace interface shall have been written to the trace log. This information includes the trace attributes, the list of trace event types (with the mapping between trace event names and trace event type identifiers), and the trace status.

In addition, unspecified information shall be written to the trace log to allow detection of a valid trace log during the *posix_trace_open*( ) operation.

The *posix_trace_shutdown*( ) function shall not return until all trace events have been flushed.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

TRL The *posix_trace_create*( ) and *posix_trace_create_withlog*( ) functions store the trace stream identifier value in the object pointed to by *trid*, if successful.

ERRORS

TRL The *posix_trace_create*( ) and *posix_trace_create_withlog*( ) functions shall fail if:

[EAGAIN] No more trace streams can be started now. {TRACE_SYS_MAX} has been exceeded.

[EINTR] The operation was interrupted by a signal. No trace stream was created.

[EINVAL] One or more of the trace parameters specified by the *attr* parameter is invalid.

[ENOMEM] The implementation does not currently have sufficient memory to create the trace stream with the specified parameters.

[EPERM] The caller does not have appropriate privilege to trace the process specified by *pid*.

[ESRCH] The *pid* argument does not refer to an existing process.

TRL The *posix_trace_create_withlog*( ) function shall fail if:

[EBADF] The *file_desc* argument is not a valid file descriptor open for writing.

[EINVAL] The *file_desc* argument refers to a file with a file type that does not support the log policy associated with the trace log.

[ENOSPC] No space left on device. The device corresponding to the argument *file_desc* does not contain the space required to create this trace log.

TRL The *posix_trace_flush*( ) and *posix_trace_shutdown*( ) functions shall fail if:

[EINVAL] The value of the *trid* argument does not correspond to an active trace stream with log.

[EFBIG] The trace log file has attempted to exceed an implementation-defined maximum file size.

[ENOSPC] No space left on device.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getres( ), *exec*, *posix_trace_attr_init*( ), *posix_trace_clear*( ), *posix_trace_close*( ), *posix_trace_eventid_equal*( ), *posix_trace_eventtypelist_getnext_id*( ), *posix_trace_flush*( ), *posix_trace_get_attr*( ), *posix_trace_get_filter*( ), *posix_trace_get_status*( ), *posix_trace_getnext_event*( ), *posix_trace_open*( ), *posix_trace_rewind*( ), *posix_trace_set_filter*( ), *posix_trace_shutdown*( ), *posix_trace_start*( ), *posix_trace_timedgetnext_event*( ), *posix_trace_trid_eventid_open*( ), *posix_trace_start*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

NAME

posix_trace_event, posix_trace_eventid_open — trace functions for instrumenting application code (**TRACING**)

SYNOPSIS

TRC
```
#include <sys/types.h>
#include <trace.h>

void posix_trace_event(trace_event_id_t event_id,
    const void *restrictdata_ptr, size_t data_len);
int posix_trace_eventid_open(const char *restrict event_name,
    trace_event_id_t *restrict event_id);
```

DESCRIPTION

The *posix_trace_event*( ) function shall record the *event_id* and the user data pointed to by *data_ptr* in the trace stream into which the calling process is being traced and in which *event_id* is not filtered out. If the total size of the user trace event data represented by *data_len* is not greater than the declared maximum size for user trace event data, then the *truncation-status* attribute of the trace event recorded is POSIX_TRACE_NOT_TRUNCATED. Otherwise, the user trace event data is truncated to this declared maximum size and the *truncation-status* attribute of the trace event recorded is POSIX_TRACE_TRUNCATED_RECORD.

If there is no trace stream created for the process or if the created trace stream is not running or if the trace event specified by *event_id* is filtered out in the trace stream, the *posix_trace_event*( ) function shall have no effect.

The *posix_trace_eventid_open*( ) function shall associate a user trace event name with a trace event type identifier for the calling process. The trace event name is the string pointed to by the argument *event_name*. It shall have a maximum of {TRACE_EVENT_NAME_MAX} characters (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). The number of user trace event type identifiers that can be defined for any given process is limited by the maximum value {TRACE_USER_EVENT_MAX}, which has the minimum value {POSIX_TRACE_USER_EVENT_MAX}.

If the Trace Inherit option is not supported, the *posix_trace_eventid_open*( ) function shall associate the user trace event name pointed to by the *event_name* argument with a trace event type identifier that is unique for the traced process, and is returned in the variable pointed to by the *event_id* argument. If the user trace event name has already been mapped for the traced process, then the previously assigned trace event type identifier shall be returned. If the per-process user trace event name limit represented by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-7 (on page 529)) user trace event shall be returned.

TRI If the Trace Inherit option is supported, the *posix_trace_eventid_open*( ) function shall associate the user trace event name pointed to by the *event_name* argument with a trace event type identifier that is unique for all the processes being traced in this same trace stream, and is returned in the variable pointed to by the *event_id* argument. If the user trace event name has already been mapped for the traced processes, then the previously assigned trace event type identifier shall be returned. If the per-process user trace event name limit represented by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined POSIX_TRACE_UNNAMED_USEREVENT (Table 2-7 (on page 529)) user trace event shall be returned.

Note: The above procedure, together with the fact that multiple processes can only be traced into the same trace stream by inheritance, ensure that all the processes that are traced into a trace stream have the same mapping of trace event names to trace event type identifiers.

If there is no trace stream created, the *posix_trace_eventid_open*( ) function shall store this information for future trace streams created for this process.

RETURN VALUE

No return value is defined for the *posix_trace_event*( ) function.

Upon successful completion, the *posix_trace_eventid_open*( ) function shall return a value of zero. Otherwise, it shall return the corresponding error number. The *posix_trace_eventid_open*( ) function stores the trace event type identifier value in the object pointed to by *event_id*, if successful.

ERRORS

The *posix_trace_eventid_open*( ) function shall fail if:

[ENAMETOOLONG]
The size of the name pointed to by *event_name* argument was longer than the implementation-defined value {TRACE_EVENT_NAME_MAX}.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_start( ), *posix_trace_trid_eventid_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #123 is applied.

IEEE PASC Interpretation 1003.1 #127 is applied, correcting some editorial errors in the names of the *posix_trace_eventid_open*( ) function and the *event_id* argument.

30230 **NAME**

30231 posix_trace_eventid_equal, posix_trace_eventid_get_name, posix_trace_trid_eventid_open —
30232 manipulate trace event type identifier (**TRACING**)

30233 **SYNOPSIS**

30234 TRC
```
#include <trace.h>

int posix_trace_eventid_equal(trace_id_t trid, trace_event_id_t event1,
    trace_event_id_t event2);
int posix_trace_eventid_get_name(trace_id_t trid,
    trace_event_id_t event, char *event_name);
int posix_trace_trid_eventid_open(trace_id_t trid,
    const char *restrict event_name,
    trace_event_id_t *restrict event);
```

30243 **DESCRIPTION**

30244 The *posix_trace_eventid_equal*( ) function shall compare the trace event type identifiers *event1* and
30245 *event2* from the same trace stream or the same trace log identified by the *trid* argument. If the
30246 trace event type identifiers *event1* and *event2* are from different trace streams, the return value
30247 shall be unspecified.

30248 The *posix_trace_eventid_get_name*( ) function shall return in the argument pointed to by
30249 *event_name*, the trace event name associated with the trace event type identifier identified by the
30250 argument *event*, for the trace stream or for the trace log identified by the *trid* argument. The
30251 name of the trace event shall have a maximum of {TRACE_EVENT_NAME_MAX} characters
30252 (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). Successive calls to
30253 this function with the same trace event type identifier and the same trace stream identifier shall
30254 return the same event name.

30255 TEF The *posix_trace_trid_eventid_open*( ) function shall associate a user trace event name with a trace
30256 event type identifier for a given trace stream. The trace stream is identified by the *trid* argument,
30257 and it shall be an active trace stream. The trace event name is the string pointed to by the
30258 argument *event_name*. It shall have a maximum of {TRACE_EVENT_NAME_MAX} characters
30259 (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). The number of user
30260 trace event type identifiers that can be defined for any given process is limited by the maximum
30261 value {TRACE_USER_EVENT_MAX}, which has the minimum value
30262 {_POSIX_TRACE_USER_EVENT_MAX}.

30263 If the Trace Inherit option is not supported, the *posix_trace_trid_eventid_open*( ) function shall
30264 associate the user trace event name pointed to by the *event_name* argument with a trace event
30265 type identifier that is unique for the process being traced in the trace stream identified by the *trid*
30266 argument, and is returned in the variable pointed to by the *event* argument. If the user trace
30267 event name has already been mapped for the traced process, then the previously assigned trace
30268 event type identifier shall be returned. If the per-process user trace event name limit represented
30269 by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined
30270 POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-7 (on page 529)) user trace event shall
30271 be returned.

30272 TEF TRI If the Trace Inherit option is supported, the *posix_trace_trid_eventid_open*( ) function shall
30273 associate the user trace event name pointed to by the *event_name* argument with a trace event
30274 type identifier that is unique for all the processes being traced in the trace stream identified by
30275 the *trid* argument, and is returned in the variable pointed to by the *event* argument. If the user
30276 trace event name has already been mapped for the traced processes, then the previously
30277 assigned trace event type identifier shall be returned. If the per-process user trace event name
30278 limit represented by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined

POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-7 (on page 529)) user trace event shall be returned.

RETURN VALUE

TEF Upon successful completion, the *posix_trace_eventid_get_name*( ) and *posix_trace_trid_eventid_open*( ) functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

The *posix_trace_eventid_equal*( ) function shall return a non-zero value if *event1* and *event2* are equal; otherwise, a value of zero shall be returned. No errors are defined. If either *event1* or *event2* are not valid trace event type identifiers for the trace stream specified by *trid* or if the *trid* is invalid, the behavior shall be unspecified.

The *posix_trace_eventid_get_name*( ) function stores the trace event name value in the object pointed to by *event_name*, if successful.

TEF The *posix_trace_trid_eventid_open*( ) function stores the trace event type identifier value in the object pointed to by *event*, if successful.

ERRORS

TEF The *posix_trace_eventid_get_name*( ) and *posix_trace_trid_eventid_open*( ) functions shall fail if:

[EINVAL] The *trid* argument was not a valid trace stream identifier.

TEF The *posix_trace_trid_eventid_open*( ) function shall fail if:

TEF [ENAMETOOLONG]
The size of the name pointed to by *event_name* argument was longer than the implementation-defined value {TRACE_EVENT_NAME_MAX}.

The *posix_trace_eventid_get_name*( ) function shall fail if:

[EINVAL] The trace event type identifier *event* was not associated with any name.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_event( ), *posix_trace_getnext_event*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretations 1003.1 #123 and #129 are applied.

30316 **NAME**

30317 posix_trace_eventid_get_name — manipulate trace event type identifier (**TRACING**)

30318 **SYNOPSIS**

30319 TRC
```
#include <trace.h>

int posix_trace_eventid_get_name(trace_id_t trid,
    trace_event_id_t event, char *event_name);
```

30320
30321
30322

30323 **DESCRIPTION**

30324 Refer to *posix_trace_eventid_equal*( ).

30325 **NAME**

30326 posix_trace_eventid_open — trace functions for instrumenting application code (**TRACING**)

30327 **SYNOPSIS**

30328 TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_eventid_open(const char *restrict event_name,
    trace_event_id_t *restrict event_id);
```
30329

30330

30331

30332

30333 **DESCRIPTION**

30334 Refer to *posix_trace_event*( ).

30335 **NAME**

30336 posix_trace_eventset_add, posix_trace_eventset_del, posix_trace_eventset_empty,
30337 posix_trace_eventset_fill, posix_trace_eventset_ismember — manipulate trace event type sets
30338 (**TRACING**)

30339 **SYNOPSIS**

30340 TRC TEF
```
#include <trace.h>

int posix_trace_eventset_add(trace_event_id_t event_id,
    trace_event_set_t *set);
int posix_trace_eventset_del(trace_event_id_t event_id,
    trace_event_set_t *set);
int posix_trace_eventset_empty(trace_event_set_t *set);
int posix_trace_eventset_fill(trace_event_set_t *set, int what);
int posix_trace_eventset_ismember(trace_event_id_t event_id,
    const trace_event_set_t *restrict set,
    int *restrict ismember);
```
30350

30351 **DESCRIPTION**

30352 These primitives manipulate sets of trace event types. They operate on data objects addressable
30353 by the application, not on the current trace event filter of any trace stream.

30354 The *posix_trace_eventset_add*( ) and *posix_trace_eventset_del*( ) functions, respectively, shall add or
30355 delete the individual trace event type specified by the value of the argument *event_id* to or from
30356 the trace event type set pointed to by the argument *set*. Adding a trace event type already in the
30357 set or deleting a trace event type not in the set shall not be considered an error.

30358 The *posix_trace_eventset_empty*( ) function shall initialize the trace event type set pointed to by
30359 the *set* argument such that all trace event types defined, both system and user, shall be excluded
30360 from the set.

30361 The *posix_trace_eventset_fill*( ) function shall initialize the trace event type set pointed to by the
30362 argument *set*, such that the set of trace event types defined by the argument *what* shall be
30363 included in the set. The value of the argument *what* shall consist of one of the following values,
30364 as defined in the **<trace.h>** header:

30365 POSIX_TRACE_WOPID_EVENTS
30366 All the process-independent implementation-defined system trace event types are included
30367 in the set.

30368 POSIX_TRACE_SYSTEM_EVENTS All the implementation-defined system trace event types are
30369 included in the set, as are those defined in IEEE Std 1003.1-200x.

30370 POSIX_TRACE_ALL_EVENTS All trace event types defined, both system and user, are included
30371 in the set.

30372 Applications shall call either *posix_trace_eventset_empty*( ) or *posix_trace_eventset_fill*( ) at least
30373 once for each object of type **trace_event_set_t** prior to any other use of that object. If such an
30374 object is not initialized in this way, but is nonetheless supplied as an argument to any of the
30375 *posix_trace_eventset_add*( ), *posix_trace_eventset_del*( ), or *posix_trace_eventset_ismember*( ) functions,
30376 the results are undefined.

30377 The *posix_trace_eventset_ismember*( ) function shall test whether the trace event type specified by
30378 the value of the argument *event_id* is a member of the set pointed to by the argument *set*. The
30379 value returned in the object pointed to by *ismember* argument is zero if the trace event type
30380 identifier is not a member of the set and a value different from zero if it is a member of the set. |

30381 **RETURN VALUE**

30382 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30383 return the corresponding error number.

30384 **ERRORS**

30385 These functions may fail if:

30386 [EINVAL] The value of one of the arguments is invalid.

30387 **EXAMPLES**

30388 None.

30389 **APPLICATION USAGE**

30390 None.

30391 **RATIONALE**

30392 None.

30393 **FUTURE DIRECTIONS**

30394 None.

30395 **SEE ALSO**

30396 *posix_trace_set_filter*( ), *posix_trace_trid_eventid_open*( ), the Base Definitions volume of
30397 IEEE Std 1003.1-200x, **<trace.h>**

30398 **CHANGE HISTORY**

30399 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30400 **NAME**

30401 posix_trace_eventtypelist_getnext_id, posix_trace_eventtypelist_rewind — iterate over a
30402 mapping of trace event types (**TRACING**)

30403 **SYNOPSIS**

30404 TRC
```
#include <trace.h>

int posix_trace_eventtypelist_getnext_id(trace_id_t trid,
    trace_event_id_t *restrict event, int *restrict unavailable);
int posix_trace_eventtypelist_rewind(trace_id_t trid);
```
30408

30409 **DESCRIPTION**

30410 The first time *posix_trace_eventtypelist_getnext_id*( ) is called, the function shall return in the
30411 variable pointed to by *event* the first trace event type identifier of the list of trace events of the
30412 trace stream identified by the *trid* argument. Successive calls to
30413 *posix_trace_eventtypelist_getnext_id*( ) return in the variable pointed to by *event* the next trace
30414 event type identifier in that same list. Each time a trace event type identifier is successfully
30415 written into the variable pointed to by the *event* argument, the variable pointed to by the
30416 *unavailable* argument shall be set to zero. When no more trace event type identifiers are
30417 available, and so none is returned, the variable pointed to by the *unavailable* argument shall be
30418 set to a value different from zero.

30419 The *posix_trace_eventtypelist_rewind*( ) function shall reset the next trace event type identifier to
30420 be read to the first trace event type identifier from the list of trace events used in the trace stream
30421 identified by *trid*.

30422 **RETURN VALUE**

30423 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30424 return the corresponding error number.

30425 The *posix_trace_eventtypelist_getnext_id*( ) function stores the trace event type identifier value in
30426 the object pointed to by *event*, if successful.

30427 **ERRORS**

30428 These functions shall fail if:

30429 [EINVAL] The *trid* argument was not a valid trace stream identifier.

30430 **EXAMPLES**

30431 None.

30432 **APPLICATION USAGE**

30433 None.

30434 **RATIONALE**

30435 None.

30436 **FUTURE DIRECTIONS**

30437 None.

30438 **SEE ALSO**

30439 *posix_trace_event*( ), *posix_trace_getnext_event*( ), *posix_trace_trid_eventid_open*( ), the Base
30440 Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

30441 **CHANGE HISTORY**

30442 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30443 IEEE PASC Interpretations 1003.1 #123 and #129 are applied.

30444 **NAME**

30445 posix_trace_flush — trace stream flush from a process (**TRACING**)

30446 **SYNOPSIS**

30447 TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_flush(trace_id_t trid);
```
30448

30449

30450

30451 **DESCRIPTION**

30452 Refer to *posix_trace_create*( ).

30453 **NAME**

30454 posix_trace_get_attr, posix_trace_get_status — retrieve the trace attributes or trace statuses
30455 (**TRACING**)

30456 **SYNOPSIS**

30457 TRC
```
#include <trace.h>

int posix_trace_get_attr(trace_id_t trid, trace_attr_t *attr);
int posix_trace_get_status(trace_id_t trid,
    struct posix_trace_status_info *statusinfo);
```
30461

30462 **DESCRIPTION**

30463 The *posix_trace_get_attr*( ) function shall copy the attributes of the active trace stream identified
30464 TRL by *trid* into the object pointed to by the *attr* argument. If the Trace Log option is supported, *trid*
30465 may represent a pre-recorded trace log.

30466 The *posix_trace_get_status*( ) function shall return, in the structure pointed to by the *statusinfo*
30467 argument, the current trace status for the trace stream identified by the *trid* argument. These
30468 status values returned in the structure pointed to by *statusinfo* shall have been appropriately
30469 TRL read to ensure that the returned values are consistent. If the Trace Log option is supported and
30470 the *trid* argument refers to a pre-recorded trace stream, the status shall be the status of the
30471 completed trace stream.

30472 Each time the *posix_trace_get_status*( ) function is used, the overrun status of the trace stream
30473 TRL shall be reset to POSIX_TRACE_NO_OVERRUN immediately after the call completes. If the
30474 Trace Log option is supported, the *posix_trace_get_status*( ) function shall behave the same as
30475 when the option is not supported except for the following differences:

30476 • If the *trid* argument refers to a trace stream with log, each time the *posix_trace_get_status*( )
30477 function is used, the log overrun status of the trace stream shall be reset to
30478 POSIX_TRACE_NO_OVERRUN and the *flush_error* status shall be reset to zero immediately
30479 after the call completes.

30480 • If the *trid* argument refers to a pre-recorded trace stream, the status returned shall be the
30481 status of the completed trace stream and the status values of the trace stream shall not be
30482 reset.
30483

30484 **RETURN VALUE**

30485 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30486 return the corresponding error number.

30487 The *posix_trace_get_attr*( ) function stores the trace attributes in the object pointed to by *attr*, if
30488 successful.

30489 The *posix_trace_get_status*( ) function stores the trace status in the object pointed to by *statusinfo*,
30490 if successful.

30491 **ERRORS**

30492 These functions shall fail if:

30493 [EINVAL] The trace stream argument *trid* does not correspond to a valid active trace
30494 stream or a valid trace log.

30495 **EXAMPLES**

30496 None.

30497 **APPLICATION USAGE**

30498 None.

30499 **RATIONALE**

30500 None.

30501 **FUTURE DIRECTIONS**

30502 None.

30503 **SEE ALSO**

30504 *posix_trace_attr_destroy*( ), *posix_trace_attr_init*( ), *posix_trace_create*( ), *posix_trace_open*( ), the Base
30505 Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

30506 **CHANGE HISTORY**

30507 First released in Issue 6. Derived from IEEE Std 1003.1q-2000. |

30508 IEEE PASC Interpretation 1003.1 #123 is applied. |

30509 **NAME**

30510 posix_trace_get_filter, posix_trace_set_filter — retrieve and set filter of an initialized trace
30511 stream (**TRACING**)

30512 **SYNOPSIS**

30513 TRC TEF
```
#include <trace.h>

int posix_trace_get_filter(trace_id_t trid, trace_event_set_t *set);
int posix_trace_set_filter(trace_id_t trid,
    const trace_event_set_t *set, int how);
```
30517

30518 **DESCRIPTION**

30519 The *posix_trace_get_filter*( ) function shall retrieve, into the argument pointed to by *set*, the actual |
30520 trace event filter from the trace stream specified by *trid*.

30521 The *posix_trace_set_filter*( ) function shall change the set of filtered trace event types after a trace |
30522 stream identified by the *trid* argument is created. This function may be called prior to starting |
30523 the trace stream, or while the trace stream is active. By default, if no call is made to
30524 *posix_trace_set_filter*( ), all trace events shall be recorded (that is, none of the trace event types are
30525 filtered out).

30526 If this function is called while the trace is in progress, a special system trace event,
30527 POSIX_TRACE_FILTER, shall be recorded in the trace indicating both the old and the new sets
30528 of filtered trace event types (see Table 2-4 (on page 528) and Table 2-6 (on page 529)).

30529 If the *posix_trace_set_filter*( ) function is interrupted by a signal, an error shall be returned and the |
30530 filter shall not be changed. In this case, the state of the trace stream shall not be changed. |

30531 The value of the argument *how* indicates the manner in which the set is to be changed and shall
30532 have one of the following values, as defined in the **<trace.h>** header:

30533 POSIX_TRACE_SET_EVENTSET
30534 The resulting set of trace event types to be filtered shall be the trace event type set pointed
30535 to by the argument *set*.

30536 POSIX_TRACE_ADD_EVENTSET
30537 The resulting set of trace event types to be filtered shall be the union of the current set and
30538 the trace event type set pointed to by the argument *set*.

30539 POSIX_TRACE_SUB_EVENTSET
30540 The resulting set of trace event types to be filtered shall be all trace event types in the
30541 current set that are not in the set pointed to by the argument *set*; that is, remove each
30542 element of the specified set from the current filter.

30543 **RETURN VALUE**

30544 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30545 return the corresponding error number.

30546 The *posix_trace_get_filter*( ) function stores the set of filtered trace event types in *set*, if successful.

30547 **ERRORS**

30548 These functions shall fail if: |

30549 [EINVAL] The value of the *trid* argument does not correspond to an active trace stream
30550 or the value of the argument pointed to by *set* is invalid.

30551 [EINTR] The operation was interrupted by a signal.

30552 **EXAMPLES**

30553 None.

30554 **APPLICATION USAGE**

30555 None.

30556 **RATIONALE**

30557 None.

30558 **FUTURE DIRECTIONS**

30559 None.

30560 **SEE ALSO**

30561 *posix_trace_eventset_add*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

30562 **CHANGE HISTORY**

30563 First released in Issue 6. Derived from IEEE Std 1003.1q-2000. |

30564 IEEE PASC Interpretation 1003.1 #123 is applied. |

30565 **NAME**

30566 posix_trace_get_status — retrieve the trace statuses (**TRACING**)

30567 **SYNOPSIS**

30568 TRC
```
#include <trace.h>

int posix_trace_get_status(trace_id_t trid,
    struct posix_trace_status_info *statusinfo);
```
30571

30572 **DESCRIPTION**

30573 Refer to *posix_trace_get_attr*( ).

NAME

posix_trace_getnext_event, posix_trace_timedgetnext_event, posix_trace_trygetnext_event — retrieve a trace event (**TRACING**)

SYNOPSIS

TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_getnext_event(trace_id_t trid,
    struct posix_trace_event_info *restrict event,
    void *restrict data, size_t num_bytes,
    size_t *restrict data_len, int *restrict unavailable);
```
TRC TMO
```
int posix_trace_timedgetnext_event(trace_id_t trid,
    struct posix_trace_event_info *restrict event,
    void *restrict data, size_t num_bytes,
    size_t *restrict data_len, int *restrict unavailable,
    const struct timespec *restrict abs_timeout);
```
TRC
```
int posix_trace_trygetnext_event(trace_id_t trid,
    struct posix_trace_event_info *restrict event,
    void *restrict data, size_t num_bytes,
    size_t *restrict data_len, int *restrict unavailable);
```

DESCRIPTION

The *posix_trace_getnext_event*( ) function shall report a recorded trace event either from an active trace stream without log TRL or a pre-recorded trace stream identified by the *trid* argument. The *posix_trace_trygetnext_event*( ) function shall report a recorded trace event from an active trace stream without log identified by the *trid* argument.

The trace event information associated with the recorded trace event shall be copied by the function into the structure pointed to by the argument *event* and the data associated with the trace event shall be copied into the buffer pointed to by the *data* argument.

The *posix_trace_getnext_event*( ) function shall block if the *trid* argument identifies an active trace stream and there is currently no trace event ready to be retrieved. When returning, if a recorded trace event was reported, the variable pointed to by the *unavailable* argument shall be set to zero. Otherwise, the variable pointed to by the *unavailable* argument shall be set to a value different from zero.

TMO The *posix_trace_timedgetnext_event*( ) function shall attempt to get another trace event from an active trace stream without log, as in the *posix_trace_getnext_event*( ) function. However, if no trace event is available from the trace stream, the implied wait shall be terminated when the timeout specified by the argument *abs_timeout* expires, and the function shall return the error [ETIMEDOUT].

The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by the clock upon which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already passed at the time of the call.

TMO TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout shall be based on the system clock as returned by the *time*( ) function. The resolution of the timeout shall be the resolution of the clock on which it is based. The **timespec** data type is defined in the **<time.h>** header.

TMO Under no circumstance shall the function fail with a timeout if a trace event is immediately available from the trace stream. The validity of the *abs_timeout* argument need not be checked if

a trace event is immediately available from the trace stream.

The behavior of this function for a pre-recorded trace stream is unspecified.

TRL The *posix_trace_trygetnext_event*( ) function shall not block. This function shall return an error if the *trid* argument identifies a pre-recorded trace stream. If a recorded trace event was reported, the variable pointed to by the *unavailable* argument shall be set to zero. Otherwise, if no trace event was reported, the variable pointed to by the *unavailable* argument shall be set to a value different from zero.

The argument *num_bytes* shall be the size of the buffer pointed to by the *data* argument. The argument *data_len* reports to the application the length in bytes of the data record just transferred. If *num_bytes* is greater than or equal to the size of the data associated with the trace event pointed to by the *event* argument, all the recorded data shall be transferred. In this case, the *truncation-status* member of the trace event structure shall be either POSIX_TRACE_NOT_TRUNCATED, if the trace event data was recorded without truncation while tracing, or POSIX_TRACE_TRUNCATED_RECORD, if the trace event data was truncated when it was recorded. If the *num_bytes* argument is less than the length of recorded trace event data, the data transferred shall be truncated to a length of *num_bytes*, the value stored in the variable pointed to by *data_len* shall be equal to *num_bytes*, and the *truncation-status* member of the *event* structure argument shall be set to POSIX_TRACE_TRUNCATED_READ (see the *posix_trace_event_info*( ) function).

The report of a trace event shall be sequential starting from the oldest recorded trace event. Trace events shall be reported in the order in which they were generated, up to an implementation-defined time resolution that causes the ordering of trace events occurring very close to each other to be unknown. Once reported, a trace event cannot be reported again from an active trace stream. Once a trace event is reported from an active trace stream without log, the trace stream shall make the resources associated with that trace event available to record future generated trace events.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

If successful, these functions store:

- The recorded trace event in the object pointed to by *event*
- The trace event information associated with the recorded trace event in the object pointed to by *data*
- The length of this trace event information in the object pointed to by *data_len*
- The value of zero in the object pointed to by *unavailable*

ERRORS

These functions shall fail if: |

[EINVAL] The trace stream identifier argument *trid* is invalid.

The *posix_trace_getnext_event*( ) and *posix_trace_timedgetnext_event*( ) functions shall fail if: |

[EINTR] The operation was interrupted by a signal, and so the call had no effect.

The *posix_trace_trygetnext_event*( ) function shall fail if: |

[EINVAL] The trace stream identifier argument *trid* does not correspond to an active trace stream.

30665 TMO The *posix_trace_timedgetnext_event*( ) function shall fail if:

30666 [EINVAL] There is no trace event immediately available from the trace stream, and the
30667 *timeout* argument is invalid.

30668 [ETIMEDOUT] No trace event was available from the trace stream before the specified
30669 timeout *timeout* expired.
30670

30671 **EXAMPLES**

30672 None.

30673 **APPLICATION USAGE**

30674 None.

30675 **RATIONALE**

30676 None.

30677 **FUTURE DIRECTIONS**

30678 None.

30679 **SEE ALSO**

30680 *posix_trace_create*( ), **posix_trace_event_info Structure**, *posix_trace_open*( ), the Base Definitions
30681 volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

30682 **CHANGE HISTORY**

30683 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30684 IEEE PASC Interpretation 1003.1 #123 is applied.

30685 **NAME**

30686 posix_trace_open — trace log management (**TRACING**)

30687 **SYNOPSIS**

30688 TCT TRL
```
#include <trace.h>

int posix_trace_open(int file_desc, trace_id_t *trid);
```
30689

30690

30691 **DESCRIPTION**

30692 Refer to *posix_trace_close*( ).

30693 **NAME**

30694 posix_trace_rewind — trace log management (**TRACING**)

30695 **SYNOPSIS**

30696 TCT TRL
```
#include <trace.h>

int posix_trace_rewind(trace_id_t trid);
```
30698

30699 **DESCRIPTION**

30700 Refer to *posix_trace_close*( ).

30701 **NAME**

30702 posix_trace_set_filter — set filter of an initialized trace stream (**TRACING**)

30703 **SYNOPSIS**

30704 TRC TEF
```
#include <trace.h>

int posix_trace_set_filter(trace_id_t trid,
    const trace_event_set_t *set, int how);
```
30705

30706

30707

30708 **DESCRIPTION**

30709 Refer to *posix_trace_get_filter*( ).

30710 **NAME**

30711 posix_trace_shutdown — trace stream shutdown from a process (**TRACING**)

30712 **SYNOPSIS**

30713 TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_shutdown(trace_id_t trid);
```
30716

30717 **DESCRIPTION**

30718 Refer to *posix_trace_create*( ).

30719 **NAME**

30720 posix_trace_start, posix_trace_stop — trace start and stop (**TRACING**)

30721 **SYNOPSIS**

30722 TRC
```
#include <trace.h>

int posix_trace_start(trace_id_t trid);
int posix_trace_stop (trace_id_t trid);
```
30725

30726 **DESCRIPTION**

30727 The *posix_trace_start*( ) and *posix_trace_stop*( ) functions, respectively, shall start and stop the
30728 trace stream identified by the argument *trid*.

30729 The effect of calling the *posix_trace_start*( ) function shall be recorded in the trace stream as the
30730 POSIX_TRACE_START system trace event and the status of the trace stream shall become
30731 POSIX_TRACE_RUNNING. If the trace stream is in progress when this function is called, the
30732 POSIX_TRACE_START system trace event shall not be recorded and the trace stream shall
30733 continue to run. If the trace stream is full, the POSIX_TRACE_START system trace event shall
30734 not be recorded and the status of the trace stream shall not be changed.

30735 The effect of calling the *posix_trace_stop*( ) function shall be recorded in the trace stream as the
30736 POSIX_TRACE_STOP system trace event and the status of the trace stream shall become
30737 POSIX_TRACE_SUSPENDED. If the trace stream is suspended when this function is called, the
30738 POSIX_TRACE_STOP system trace event shall not be recorded and the trace stream shall remain
30739 suspended. If the trace stream is full, the POSIX_TRACE_STOP system trace event shall not be
30740 recorded and the status of the trace stream shall not be changed.

30741 **RETURN VALUE**

30742 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30743 return the corresponding error number.

30744 **ERRORS**

30745 These functions shall fail if:

30746 [EINVAL] The value of the argument *trid* does not correspond to an active trace stream
30747 and thus no trace stream was started or stopped.

30748 [EINTR] The operation was interrupted by a signal and thus the trace stream was not
30749 necessarily started or stopped.

30750 **EXAMPLES**

30751 None.

30752 **APPLICATION USAGE**

30753 None.

30754 **RATIONALE**

30755 None.

30756 **FUTURE DIRECTIONS**

30757 None.

30758 **SEE ALSO**

30759 *posix_trace_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

30760 **CHANGE HISTORY**

30761 First released in Issue 6. Derived from IEEE Std 1003.1q-2000. |

30762 IEEE PASC Interpretation 1003.1 #123 is applied. |

30763 **NAME**

30764 posix_trace_timedgetnext_event, — retrieve a trace event (**TRACING**)

30765 **SYNOPSIS**

30766 TRC TMO
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_timedgetnext_event(trace_id_t trid,
    struct posix_trace_event_info *restrict event,
    void *restrict data, size_t num_bytes,
    size_t *restrict data_len, int *restrict unavailable,
    const struct timespec *restrict abs_timeout);
```
30773

30774 **DESCRIPTION**

30775 Refer to *posix_trace_getnext_event*( ).

30776 **NAME**

30777 posix_trace_trid_eventid_open — manipulate trace event type identifier (**TRACING**)

30778 **SYNOPSIS**

30779 TRC TEF
```
#include <trace.h>

int posix_trace_trid_eventid_open(trace_id_t trid,
    const char *restrict event_name,
    trace_event_id_t *restrict event);
```

30783

30784 **DESCRIPTION**

30785 Refer to *posix_trace_eventid_equal*( ).

30786 **NAME**

30787 posix_trace_trygetnext_event — retrieve a trace event (**TRACING**)

30788 **SYNOPSIS**

30789 TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_trygetnext_event(trace_id_t trid,
    struct posix_trace_event_info *restrict event,
    void *restrict data, size_t num_bytes,
    size_t *restrict data_len, int *restrict unavailable);
```
30795

30796 **DESCRIPTION**

30797 Refer to *posix_trace_getnext_event*( ).

30798 **NAME**

30799 posix_typed_mem_get_info — query typed memory information (**ADVANCED REALTIME**)

30800 **SYNOPSIS**

30801 TYM
```
#include <sys/mman.h>
```

30802
```
int posix_typed_mem_get_info(int fildes,
    struct posix_typed_mem_info *info);
```

30804

30805 **DESCRIPTION**

30806 The *posix_typed_mem_get_info*( ) function shall return, in the *posix_tmi_length* field of the
30807 **posix_typed_mem_info** structure pointed to by *info*, the maximum length which may be
30808 successfully allocated by the typed memory object designated by *fildes*. This maximum length
30809 shall take into account the flag POSIX_TYPED_MEM_ALLOCATE or
30810 POSIX_TYPED_MEM_ALLOCATE_CONTIG specified when the typed memory object
30811 represented by *fildes* was opened. The maximum length is dynamic; therefore, the value returned
30812 is valid only while the current mapping of the corresponding typed memory pool remains
30813 unchanged.

30814 If *fildes* represents a typed memory object opened with neither the
30815 POSIX_TYPED_MEM_ALLOCATE flag nor the POSIX_TYPED_MEM_ALLOCATE_CONTIG
30816 flag specified, the returned value of *info->posix_tmi_length* is unspecified.

30817 The *posix_typed_mem_get_info*( ) function may return additional implementation-defined
30818 information in other fields of the **posix_typed_mem_info** structure pointed to by *info*.

30819 If the memory object specified by *fildes* is not a typed memory object, then the behavior of this
30820 function is undefined.

30821 **RETURN VALUE**

30822 Upon successful completion, the *posix_typed_mem_get_info*( ) function shall return zero;
30823 otherwise, the corresponding error status value shall be returned.

30824 **ERRORS**

30825 The *posix_typed_mem_get_info*( ) function shall fail if:

30826 [EBADF] The *fildes* argument is not a valid open file descriptor.

30827 [ENODEV] The *fildes* argument is not connected to a memory object supported by this
30828 function.

30829 This function shall not return an error code of [EINTR].

30830 **EXAMPLES**

30831 None.

30832 **APPLICATION USAGE**

30833 None.

30834 **RATIONALE**

30835 An application that needs to allocate a block of typed memory with length dependent upon the
30836 amount of memory currently available must either query the typed memory object to obtain the
30837 amount available, or repeatedly invoke *mmap*( ) attempting to guess an appropriate length.
30838 While the latter method is existing practice with *malloc*( ), it is awkward and imprecise. The
30839 *posix_typed_mem_get_info*( ) function allows an application to immediately determine available
30840 memory. This is particularly important for typed memory objects that may in some cases be
30841 scarce resources. Note that when a typed memory pool is a shared resource, some form of
30842 mutual exclusion or synchronization may be required while typed memory is being queried and

allocated to prevent race conditions.

The existing *fstat*( ) function is not suitable for this purpose. We realize that implementations may wish to provide other attributes of typed memory objects (for example, alignment requirements, page size, and so on). The *fstat*( ) function returns a structure which is not extensible and, furthermore, contains substantial information that is inappropriate for typed memory objects.

FUTURE DIRECTIONS

None.

SEE ALSO

fstat( ), *mmap*( ), *posix_typed_mem_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

30856 **NAME**

30857 posix_typed_mem_open — open a typed memory object (**ADVANCED REALTIME**)

30858 **SYNOPSIS**

30859 TYM
```
#include <sys/mman.h>
```

30860
```
int posix_typed_mem_open(const char *name, int oflag, int tflag);
```

30861

30862 **DESCRIPTION**

30863 The *posix_typed_mem_open*( ) function shall establish a connection between the typed memory
30864 object specified by the string pointed to by *name* and a file descriptor. It shall create an open file
30865 description that refers to the typed memory object and a file descriptor that refers to that open
30866 file description. The file descriptor is used by other functions to refer to that typed memory
30867 object. It is unspecified whether the name appears in the file system and is visible to other |
30868 functions that take pathnames as arguments. The *name* argument shall conform to the |
30869 construction rules for a pathname. If *name* begins with the slash character, then processes calling |
30870 *posix_typed_mem_open*( ) with the same value of *name* shall refer to the same typed memory
30871 object. If *name* does not begin with the slash character, the effect is implementation-defined. The
30872 interpretation of slash characters other than the leading slash character in *name* is
30873 implementation-defined.

30874 Each typed memory object supported in a system shall be identified by a name which specifies |
30875 not only its associated typed memory pool, but also the path or port by which it is accessed. That
30876 is, the same typed memory pool accessed via several different ports shall have several different |
30877 corresponding names. The binding between names and typed memory objects is established in
30878 an implementation-defined manner. Unlike shared memory objects, there is no way within |
30879 IEEE Std 1003.1-200x for a program to create a typed memory object. |

30880 The value of *tflag* shall determine how the typed memory object behaves when subsequently |
30881 mapped by calls to *mmap*( ). At most, one of the following flags defined in **<sys/mman.h>** may |
30882 be specified:

30883 POSIX_TYPED_MEM_ALLOCATE
30884 Allocate on *mmap*( ).

30885 POSIX_TYPED_MEM_ALLOCATE_CONTIG
30886 Allocate contiguously on *mmap*( ).

30887 POSIX_TYPED_MEM_MAP_ALLOCATABLE
30888 Map on *mmap*( ), without affecting allocatability.

30889 If *tflag* has the flag POSIX_TYPED_MEM_ALLOCATE specified, any subsequent call to *mmap*( )
30890 using the returned file descriptor shall result in allocation and mapping of typed memory from
30891 the specified typed memory pool. The allocated memory may be a contiguous previously
30892 unallocated area of the typed memory pool or several non-contiguous previously unallocated
30893 areas (mapped to a contiguous portion of the process address space). If *tflag* has the flag
30894 POSIX_TYPED_MEM_ALLOCATE_CONTIG specified, any subsequent call to *mmap*( ) using the
30895 returned file descriptor shall result in allocation and mapping of a single contiguous previously
30896 unallocated area of the typed memory pool (also mapped to a contiguous portion of the process
30897 address space). If *tflag* has none of the flags POSIX_TYPED_MEM_ALLOCATE or
30898 POSIX_TYPED_MEM_ALLOCATE_CONTIG specified, any subsequent call to *mmap*( ) using the
30899 returned file descriptor shall map an application-chosen area from the specified typed memory
30900 pool such that this mapped area becomes unavailable for allocation until unmapped by all
30901 processes. If *tflag* has the flag POSIX_TYPED_MEM_MAP_ALLOCATABLE specified, any
30902 subsequent call to *mmap*( ) using the returned file descriptor shall map an application-chosen
30903 area from the specified typed memory pool without an effect on the availability of that area for

allocation; that is, mapping such an object leaves each byte of the mapped area unallocated if it was unallocated prior to the mapping or allocated if it was allocated prior to the mapping. The appropriate privilege to specify the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag is implementation-defined.

If successful, *posix_typed_mem_open*( ) shall return a file descriptor for the typed memory object that is the lowest numbered file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor shall not share it with any other processes. It is unspecified whether the file offset is set. The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared.

The behavior of *msync*( ), *ftruncate*( ), and all file operations other than *mmap*( ), *posix_mem_offset*( ), *posix_typed_mem_get_info*( ), *fstat*( ), *dup*( ), *dup2*( ), and *close*( ), is unspecified when passed a file descriptor connected to a typed memory object by this function.

The file status flags of the open file description shall be set according to the value of *oflag*. Applications shall specify exactly one of the three access mode values described below and defined in the **<fcntl.h>** header, as the value of *oflag*.

O_RDONLY Open for read access only.

O_WRONLY Open for write access only.

O_RDWR Open for read or write access.

RETURN VALUE

Upon successful completion, the *posix_typed_mem_open*( ) function shall return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *posix_typed_mem_open*( ) function shall fail if:

[EACCES] The typed memory object exists and the permissions specified by *oflag* are denied.

[EINTR] The *posix_typed_mem_open*( ) operation was interrupted by a signal.

[EINVAL] The flags specified in *tflag* are invalid (more than one of POSIX_TYPED_MEM_ALLOCATE, POSIX_TYPED_MEM_ALLOCATE_CONTIG, or POSIX_TYPED_MEM_MAP_ALLOCATABLE is specified).

[EMFILE] Too many file descriptors are currently in use by this process.

[ENAMETOOLONG] The length of the *name* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

[ENFILE] Too many file descriptors are currently open in the system.

[ENOENT] The named typed memory object does not exist.

[EPERM] The caller lacks the appropriate privilege to specify the flag POSIX_TYPED_MEM_MAP_ALLOCATABLE in argument *tflag*.

30943 **EXAMPLES**

30944 None.

30945 **APPLICATION USAGE**

30946 None.

30947 **RATIONALE**

30948 None.

30949 **FUTURE DIRECTIONS**

30950 None.

30951 **SEE ALSO**

30952 *close*( ), *dup*( ), *exec*, *fcntl*( ), *fstat*( ), *ftruncate*( ), *mmap*( ), *msync*( ), *posix_mem_offset*( ),
30953 *posix_typed_mem_get_info*( ), *umask*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
30954 **<fcntl.h,> <sys/mman.h>**

30955 **CHANGE HISTORY**

30956 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

30957 **NAME**

30958 pow, powf, powl — power function

30959 **SYNOPSIS**

30960
```
#include <math.h>
```

30961
```
double pow(double x, double y);
```
30962
```
float powf(float x, float y);
```
30963
```
long double powl(long double x, long double y);
```

30964 **DESCRIPTION**

30965 CX The functionality described on this reference page is aligned with the ISO C standard. Any
30966 conflict between the requirements described here and the ISO C standard is unintentional. This
30967 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

30968 These functions shall compute the value of x raised to the power y, x^y. If x is negative, the
30969 application shall ensure that y is an integer value.

30970 An application wishing to check for error situations should set *errno* to zero and call
30971 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
30972 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
30973 zero, an error has occurred.

30974 **RETURN VALUE**

30975 Upon successful completion, these functions shall return the value of x raised to the power y.

30976 MX For finite values of $x < 0$, and finite non-integer values of y, a domain error shall occur and either
30977 a NaN (if representable), or an implementation-defined value shall be returned.

30978 If the correct value would cause overflow, a range error shall occur and *pow*( ), *powf*( ), and
30979 *powl*( ) shall return HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

30980 If the correct value would cause underflow, and is not representable, a range error may occur,
30981 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

30982 MX If x or y is a NaN, a NaN shall be returned (unless specified elsewhere in this description).

30983 For any value of y (including NaN), if x is +1, 1.0 shall be returned.

30984 For any value of x (including NaN), if y is ±0, 1.0 shall be returned.

30985 For any odd integer value of $y > 0$, if x is ±0, ±0 shall be returned.

30986 For $y > 0$ and not an odd integer, if x is ±0, +0 shall be returned.

30987 If x is –1, and y is ±Inf, 1.0 shall be returned.

30988 For $|x| < 1$, if y is –Inf, +Inf shall be returned.

30989 For $|x| > 1$, if y is –Inf, +0 shall be returned.

30990 For $|x| < 1$, if y is +Inf, +0 shall be returned.

30991 For $|x| > 1$, if y is +Inf, +Inf shall be returned.

30992 For y an odd integer < 0, if x is –Inf, –0 shall be returned.

30993 For $y < 0$ and not an odd integer, if x is –Inf, +0 shall be returned.

30994 For y an odd integer > 0, if x is –Inf, –Inf shall be returned.

30995 For $y > 0$ and not an odd integer, if x is –Inf, +Inf shall be returned.

For $y < 0$, if x is +Inf, +0 shall be returned.

For $y > 0$, if x is +Inf, +Inf shall be returned.

For y an odd integer < 0, if x is ±0, a pole error shall occur and ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL shall be returned for *pow*( ), *powf*( ), and *powl*( ), respectively.

For $y < 0$ and not an odd integer, if x is ±0, a pole error shall occur and HUGE_VAL, HUGE_VALF, and HUGE_VALL shall be returned for *pow*( ), *powf*( ), and *powl*( ), respectively.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

ERRORS

These functions shall fail if:

Domain Error — The value of x is negative and y is a finite non-integer.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

MX Pole Error — The value of x is zero and y is negative.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised.

Range Error — The result overflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

Range Error — The result underflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exp( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *powf*( ) and *powl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

31051 **NAME**

31052 pread — read from a file

31053 **SYNOPSIS**

31054 XSI
```
#include <unistd.h>
```

31055
```
ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);
```

31056

31057 **DESCRIPTION**

31058 Refer to *read*( ).

31059 **NAME**

31060 printf — print formatted output

31061 **SYNOPSIS**

31062 `#include <stdio.h>`

31063 `int printf(const char *restrict format, ...);`

31064 **DESCRIPTION**

31065 Refer to *fprintf*( ).

31066 **NAME**

31067 pselect, select — synchronous I/O multiplexing

31068 **SYNOPSIS**

31069
```
#include <sys/select.h>
```

31070
```
int pselect(int nfds, fd_set *restrict readfds,
    fd_set *restrict writefds, fd_set *restrict errorfds,
    const struct timespec *restrict timeout,
    const sigset_t *restrict sigmask);
int select(int nfds, fd_set *restrict readfds,
    fd_set *restrict writefds, fd_set *restrict errorfds,
    struct timeval *restrict timeout);
void FD_CLR(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

31081 **DESCRIPTION**

31082 The *pselect*( ) function shall examine the file descriptor sets whose addresses are passed in the
31083 *readfds*, *writefds*, and *errorfds* parameters to see if some of their descriptors are ready for reading,
31084 are ready for writing, or have an exceptional condition pending, respectively.

31085 The *select*( ) function shall be equivalent to the *pselect*( ) function, except as follows:

31086 • For the *select*( ) function, the timeout period is given in seconds and microseconds in an
31087 argument of type **struct timeval**, whereas for the *pselect*( ) function the timeout period is
31088 given in seconds and nanoseconds in an argument of type **struct timespec**.

31089 • The *select*( ) function has no *sigmask* argument; it shall behave as *pselect*( ) does when *sigmask*
31090 is a null pointer.

31091 • Upon successful completion, the *select*( ) function may modify the object pointed to by the
31092 *timeout* argument.

31093 The *pselect*( ) and *select*( ) functions shall support regular files, terminal and pseudo-terminal
31094 XSR devices, STREAMS-based files, FIFOs, pipes, and sockets. The behavior of *pselect*( ) and *select*( )
31095 on file descriptors that refer to other types of file is unspecified.

31096 The *nfds* argument specifies the range of descriptors to be tested. The first *nfds* descriptors shall
31097 be checked in each set; that is, the descriptors from zero through *nfds*−1 in the descriptor sets
31098 shall be examined.

31099 If the *readfds* argument is not a null pointer, it points to an object of type **fd_set** that on input
31100 specifies the file descriptors to be checked for being ready to read, and on output indicates
31101 which file descriptors are ready to read.

31102 If the *writefds* argument is not a null pointer, it points to an object of type **fd_set** that on input
31103 specifies the file descriptors to be checked for being ready to write, and on output indicates
31104 which file descriptors are ready to write.

31105 If the *errorfds* argument is not a null pointer, it points to an object of type **fd_set** that on input
31106 specifies the file descriptors to be checked for error conditions pending, and on output indicates
31107 which file descriptors have error conditions pending.

31108 Upon successful completion, the *pselect*( ) or *select*( ) function shall modify the objects pointed to
31109 by the *readfds*, *writefds*, and *errorfds* arguments to indicate which file descriptors are ready for
31110 reading, ready for writing, or have an error condition pending, respectively, and shall return the
31111 total number of ready descriptors in all the output sets. For each file descriptor less than *nfds*, the

corresponding bit shall be set on successful completion if it was set on input and the associated condition is true for that file descriptor.

If none of the selected descriptors are ready for the requested operation, the *pselect*( ) or *select*( ) function shall block until at least one of the requested operations becomes ready, until the *timeout* occurs, or until interrupted by a signal. The *timeout* parameter controls how long the *pselect*( ) or *select*( ) function shall take before timing out. If the *timeout* parameter is not a null pointer, it specifies a maximum interval to wait for the selection to complete. If the specified time interval expires without any requested operation becoming ready, the function shall return. If the *timeout* parameter is a null pointer, then the call to *pselect*( ) or *select*( ) shall block indefinitely until at least one descriptor meets the specified criteria. To effect a poll, the *timeout* parameter should not be a null pointer, and should point to a zero-valued **timespec** structure.

The use of a timeout does not affect any pending timers set up by *alarm*( ), *ualarm*( ), or *setitimer*( ).

Implementations may place limitations on the maximum timeout interval supported. All implementations shall support a maximum timeout interval of at least 31 days. If the *timeout* argument specifies a timeout interval greater than the implementation-defined maximum value, the maximum value shall be used as the actual timeout value. Implementations may also place limitations on the granularity of timeout intervals. If the requested timeout interval requires a finer granularity than the implementation supports, the actual timeout interval shall be rounded up to the next supported value.

If *sigmask* is not a null pointer, then the *pselect*( ) function shall replace the signal mask of the process by the set of signals pointed to by *sigmask* before examining the descriptors, and shall restore the signal mask of the process before returning.

A descriptor shall be considered ready for reading when a call to an input function with O_NONBLOCK clear would not block, whether or not the function would transfer data successfully. (The function might return data, an end-of-file indication, or an error other than one indicating that it is blocked, and in each of these cases the descriptor shall be considered ready for reading.)

A descriptor shall be considered ready for writing when a call to an output function with O_NONBLOCK clear would not block, whether or not the function would transfer data successfully.

If a socket has a pending error, it shall be considered to have an exceptional condition pending. Otherwise, what constitutes an exceptional condition is file type-specific. For a file descriptor for use with a socket, it is protocol-specific except as noted below. For other file types it is implementation-defined. If the operation is meaningless for a particular file type, *pselect*( ) or *select*( ) shall indicate that the descriptor is ready for read or write operations, and shall indicate that the descriptor has no exceptional condition pending.

If a descriptor refers to a socket, the implied input function is the *recvmsg*( ) function with parameters requesting normal and ancillary data, such that the presence of either type shall cause the socket to be marked as readable. The presence of out-of-band data shall be checked if the socket option SO_OOBINLINE has been enabled, as out-of-band data is enqueued with normal data. If the socket is currently listening, then it shall be marked as readable if an incoming connection request has been received, and a call to the *accept*( ) function shall complete without blocking.

If a descriptor refers to a socket, the implied output function is the *sendmsg*( ) function supplying an amount of normal data equal to the current value of the SO_SNDLOWAT option for the socket. If a non-blocking call to the *connect*( ) function has been made for a socket, and the connection attempt has either succeeded or failed leaving a pending error, the socket shall be

marked as writable.

A socket shall be considered to have an exceptional condition pending if a receive operation with O_NONBLOCK clear for the open file description and with the MSG_OOB flag set would return out-of-band data without blocking. (It is protocol-specific whether the MSG_OOB flag would be used to read out-of-band data.) A socket shall also be considered to have an exceptional condition pending if an out-of-band data mark is present in the receive queue. Other circumstances under which a socket may be considered to have an exceptional condition pending are protocol-specific and implementation-defined.

If the *readfds*, *writefds*, and *errorfds* arguments are all null pointers and the *timeout* argument is not a null pointer, the *pselect*( ) or *select*( ) function shall block for the time specified, or until interrupted by a signal. If the *readfds*, *writefds*, and *errorfds* arguments are all null pointers and the *timeout* argument is a null pointer, the *pselect*( ) or *select*( ) function shall block until interrupted by a signal.

File descriptors associated with regular files shall always select true for ready to read, ready to write, and error conditions.

On failure, the objects pointed to by the *readfds*, *writefds*, and *errorfds* arguments shall not be modified. If the timeout interval expires without the specified condition being true for any of the specified file descriptors, the objects pointed to by the *readfds*, *writefds*, and *errorfds* arguments shall have all bits set to 0.

File descriptor masks of type **fd_set** can be initialized and tested with *FD_CLR*( ), *FD_ISSET*( ), *FD_SET*( ), and *FD_ZERO*( ). It is unspecified whether each of these is a macro or a function. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with any of these names, the behavior is undefined.

FD_CLR(*fd*, *fdsetp*) shall remove the file descriptor *fd* from the set pointed to by *fdsetp*. If *fd* is not a member of this set, there shall be no effect on the set, nor will an error be returned.

FD_ISSET(*fd*, *fdsetp*) shall evaluate to non-zero if the file descriptor *fd* is a member of the set pointed to by *fdsetp*, and shall evaluate to zero otherwise.

FD_SET(*fd*, *fdsetp*) shall add the file descriptor *fd* to the set pointed to by *fdsetp*. If the file descriptor *fd* is already in this set, there shall be no effect on the set, nor will an error be returned.

FD_ZERO(*fdsetp*) shall initialize the descriptor set pointed to by *fdsetp* to the null set. No error is returned if the set is not empty at the time *FD_ZERO*( ) is invoked.

The behavior of these macros is undefined if the *fd* argument is less than 0 or greater than or equal to FD_SETSIZE, or if *fd* is not a valid file descriptor, or if any of the arguments are expressions with side effects.

RETURN VALUE

Upon successful completion, the *pselect*( ) and *select*( ) functions shall return the total number of bits set in the bit masks. Otherwise, –1 shall be returned, and shall set *errno* to indicate the error.

FD_CLR( ), *FD_SET*( ), and *FD_ZERO*( ) not return a value. *FD_ISSET*( ) shall return a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.

ERRORS

Under the following conditions, *pselect*( ) and *select*( ) shall fail and set *errno* to:

[EBADF] One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor.

31204		[EINTR]	The function was interrupted before any of the selected events occurred and
31205			before the timeout interval expired.
31206	XSI		If SA_RESTART has been set for the interrupting signal, it is implementation-
31207			defined whether the function restarts or returns with [EINTR].
31208		[EINVAL]	An invalid timeout interval was specified.
31209		[EINVAL]	The *nfds* argument is less than 0 or greater than FD_SETSIZE.
31210	XSR	[EINVAL]	One of the specified file descriptors refers to a STREAM or multiplexer that is
31211			linked (directly or indirectly) downstream from a multiplexer.

31212 **EXAMPLES**

31213 None.

31214 **APPLICATION USAGE**

31215 None.

31216 **RATIONALE**

31217 In previous versions of the Single UNIX Specification, the *select*( ) function was defined in the
31218 **<sys/time.h>** header. This is now changed to **<sys/select.h>**. The rationale for this change was
31219 as follows: the introduction of the *pselect*( ) function included the **<sys/select.h>** header and the
31220 **<sys/select.h>** header defines all the related definitions for the *pselect*( ) and *select*( ) functions.
31221 Backwards-compatibility to existing XSI implementations is handled by allowing **<sys/time.h>**
31222 to include **<sys/select.h>**.

31223 **FUTURE DIRECTIONS**

31224 None.

31225 **SEE ALSO**

31226 *accept*( ), *alarm*( ), *connect*( ), *fcntl*( ), *poll*( ), *read*( ), *recvmsg*( ), *sendmsg*( ), *setitimer*( ), *ualarm*( ),
31227 *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/select.h>**, **<sys/time.h>**

31228 **CHANGE HISTORY**

31229 First released in Issue 4, Version 2.

31230 **Issue 5**

31231 Moved from X/OPEN UNIX extension to BASE.

31232 In the ERRORS section, the text has been changed to indicate that [EINVAL] is returned when
31233 *nfds* is less than 0 or greater than FD_SETSIZE. It previously stated less than 0, or greater than or
31234 equal to FD_SETSIZE.

31235 Text about *timeout* is moved from the APPLICATION USAGE section to the DESCRIPTION.

31236 **Issue 6**

31237 The Open Group Corrigendum U026/6 is applied, changing the occurrences of *readfs* and *writefs*
31238 in the *select*( ) DESCRIPTION to be *readfds* and *writefds*.

31239 Text referring to sockets is added to the DESCRIPTION.

31240 The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are
31241 marked as part of the XSI STREAMS Option Group.

31242 The following new requirements on POSIX implementations derive from alignment with the
31243 Single UNIX Specification:

31244 • These functions are now mandatory.

31245 The *pselect*( ) function is added for alignment with IEEE Std 1003.1g-2000 and additional detail
31246 related to sockets semantics is added to the DESCRIPTION.

31247 The *select*( ) function now requires inclusion of **<sys/select.h>**.

31248 The **restrict** keyword is added to the *select*( ) prototype for alignment with the
31249 ISO / IEC 9899: 1999 standard.

NAME

pthread_atfork — register fork handlers

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_atfork(void (*prepare)(void), void (*parent)(void),
    void (*child)(void));
```

DESCRIPTION

The *pthread_atfork*( ) function shall declare fork handlers to be called before and after *fork*( ), in the context of the thread that called *fork*( ). The *prepare* fork handler shall be called before *fork*( ) processing commences. The *parent* fork handle shall be called after *fork*( ) processing completes in the parent process. The *child* fork handler shall be called after *fork*( ) processing completes in the child process. If no handling is desired at one or more of these three points, the corresponding fork handler address(es) may be set to NULL.

The order of calls to *pthread_atfork*( ) is significant. The *parent* and *child* fork handlers shall be called in the order in which they were established by calls to *pthread_atfork*( ). The *prepare* fork handlers shall be called in the opposite order.

RETURN VALUE

Upon successful completion, *pthread_atfork*( ) shall return a value of zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_atfork*( ) function shall fail if:

[ENOMEM] Insufficient table space exists to record the fork handler addresses.

The *pthread_atfork*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

There are at least two serious problems with the semantics of *fork*( ) in a multi-threaded program. One problem has to do with state (for example, memory) covered by mutexes. Consider the case where one thread has a mutex locked and the state covered by that mutex is inconsistent while another thread calls *fork*( ). In the child, the mutex is in the locked state (locked by a nonexistent thread and thus can never be unlocked). Having the child simply reinitialize the mutex is unsatisfactory since this approach does not resolve the question about how to correct or otherwise deal with the inconsistent state in the child.

It is suggested that programs that use *fork*( ) call an *exec* function very soon afterwards in the child process, thus resetting all states. In the meantime, only a short list of async-signal-safe library routines are promised to be available.

Unfortunately, this solution does not address the needs of multi-threaded libraries. Application programs may not be aware that a multi-threaded library is in use, and they feel free to call any number of library routines between the *fork*( ) and *exec* calls, just as they always have. Indeed, they may be extant single-threaded programs and cannot, therefore, be expected to obey new restrictions imposed by the threads library.

On the other hand, the multi-threaded library needs a way to protect its internal state during *fork*( ) in case it is re-entered later in the child process. The problem arises especially in multi-threaded I/O libraries, which are almost sure to be invoked between the *fork*( ) and *exec* calls to effect I/O redirection. The solution may require locking mutex variables during *fork*( ), or it may entail simply resetting the state in the child after the *fork*( ) processing completes.

The *pthread_atfork*( ) function provides multi-threaded libraries with a means to protect themselves from innocent application programs that call *fork*( ), and it provides multi-threaded application programs with a standard mechanism for protecting themselves from *fork*( ) calls in a library routine or the application itself.

The expected usage is that the *prepare* handler acquires all mutex locks and the other two fork handlers release them.

For example, an application can supply a *prepare* routine that acquires the necessary mutexes the library maintains and supply *child* and *parent* routines that release those mutexes, thus ensuring that the child gets a consistent snapshot of the state of the library (and that no mutexes are left stranded). Alternatively, some libraries might be able to supply just a *child* routine that reinitializes the mutexes in the library and all associated states to some known value (for example, what it was when the image was originally executed).

When *fork*( ) is called, only the calling thread is duplicated in the child process. Synchronization variables remain in the same state in the child as they were in the parent at the time *fork*( ) was called. Thus, for example, mutex locks may be held by threads that no longer exist in the child process, and any associated states may be inconsistent. The parent process may avoid this by explicit code that acquires and releases locks critical to the child via *pthread_atfork*( ). In addition, any critical threads need to be recreated and reinitialized to the proper state in the child (also via *pthread_atfork*( )).

A higher-level package may acquire locks on its own data structures before invoking lower-level packages. Under this scenario, the order specified for fork handler calls allows a simple rule of initialization for avoiding package deadlock: a package initializes all packages on which it depends before it calls the *pthread_atfork*( ) function for itself.

FUTURE DIRECTIONS

None.

SEE ALSO

atexit( ), *fork*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 5. Derived from the POSIX Threads Extension.

IEEE PASC Interpretation 1003.1c #4 is applied.

Issue 6

The *pthread_atfork*( ) function is marked as part of the Threads option.

The **<pthread.h>** header is added to the SYNOPSIS.

31332 **NAME**

31333 pthread_attr_destroy, pthread_attr_init — destroy and initialize threads attributes object

31334 **SYNOPSIS**

31335 THR
```
#include <pthread.h>

int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_init(pthread_attr_t *attr);
```
31338

31339 **DESCRIPTION**

31340 The *pthread_attr_destroy*( ) function shall destroy a thread attributes object. An implementation | 31341 may cause *pthread_attr_destroy*( ) to set *attr* to an implementation-defined invalid value. A | 31342 destroyed *attr* attributes object can be reinitialized using *pthread_attr_init*( ); the results of | 31343 otherwise referencing the object after it has been destroyed are undefined. |

31344 The *pthread_attr_init*( ) function shall initialize a thread attributes object *attr* with the default 31345 value for all of the individual attributes used by a given implementation.

31346 The resulting attributes object (possibly modified by setting individual attribute values), when 31347 used by *pthread_create*( ) defines the attributes of the thread created. A single attributes object can 31348 be used in multiple simultaneous calls to *pthread_create*( ). Results are undefined if | 31349 *pthread_attr_init*( ) is called specifying an already initialized *attr* attributes object. |

31350 **RETURN VALUE**

31351 Upon successful completion, *pthread_attr_destroy*( ) and *pthread_attr_init*( ) shall return a value of 31352 0; otherwise, an error number shall be returned to indicate the error.

31353 **ERRORS**

31354 The *pthread_attr_init*( ) function shall fail if:

31355 [ENOMEM] Insufficient memory exists to initialize the thread attributes object.

31356 These functions shall not return an error code of [EINTR].

31357 **EXAMPLES**

31358 None.

31359 **APPLICATION USAGE**

31360 None.

31361 **RATIONALE**

31362 Attributes objects are provided for threads, mutexes, and condition variables as a mechanism to 31363 support probable future standardization in these areas without requiring that the function itself 31364 be changed.

31365 Attributes objects provide clean isolation of the configurable aspects of threads. For example, 31366 ‘‘stack size’’ is an important attribute of a thread, but it cannot be expressed portably. When 31367 porting a threaded program, stack sizes often need to be adjusted. The use of attributes objects 31368 can help by allowing the changes to be isolated in a single place, rather than being spread across 31369 every instance of thread creation.

31370 Attributes objects can be used to set up ‘‘classes’ of threads with similar attributes; for example, 31371 ‘‘threads with large stacks and high priority’’ or ‘‘threads with minimal stacks’’. These classes 31372 can be defined in a single place and then referenced wherever threads need to be created. 31373 Changes to ‘‘class’’ decisions become straightforward, and detailed analysis of each 31374 *pthread_create*( ) call is not required.

31375 The attributes objects are defined as opaque types as an aid to extensibility. If these objects had 31376 been specified as structures, adding new attributes would force recompilation of all multi-

threaded programs when the attributes objects are extended; this might not be possible if different program components were supplied by different vendors.

Additionally, opaque attributes objects present opportunities for improving performance. Argument validity can be checked once when attributes are set, rather than each time a thread is created. Implementations often need to cache kernel objects that are expensive to create. Opaque attributes objects provide an efficient mechanism to detect when cached objects become invalid due to attribute changes.

Since assignment is not necessarily defined on a given opaque type, implementation-defined default values cannot be defined in a portable way. The solution to this problem is to allow attributes objects to be initialized dynamically by attributes object initialization functions, so that default values can be supplied automatically by the implementation.

The following proposal was provided as a suggested alternative to the supplied attributes:

1. Maintain the style of passing a parameter formed by the bitwise-inclusive OR of flags to the initialization routines (*pthread_create*( ), *pthread_mutex_init*( ), *pthread_cond_init*( )). The parameter containing the flags should be an opaque type for extensibility. If no flags are set in the parameter, then the objects are created with default characteristics. An implementation may specify implementation-defined flag values and associated behavior.

2. If further specialization of mutexes and condition variables is necessary, implementations may specify additional procedures that operate on the **pthread_mutex_t** and **pthread_cond_t** objects (instead of on attributes objects).

The difficulties with this solution are:

1. A bitmask is not opaque if bits have to be set into bitvector attributes objects using explicitly-coded bitwise-inclusive OR operations. If the set of options exceeds an **int**, application programmers need to know the location of each bit. If bits are set or read by encapsulation (that is, get and set functions), then the bitmask is merely an implementation of attributes objects as currently defined and should not be exposed to the programmer.

2. Many attributes are not Boolean or very small integral values. For example, scheduling policy may be placed in 3-bit or 4-bit, but priority requires 5-bit or more, thereby taking up at least 8 bits out of a possible 16 bits on machines with 16-bit integers. Because of this, the bitmask can only reasonably control whether particular attributes are set or not, and it cannot serve as the repository of the value itself. The value needs to be specified as a function parameter (which is non-extensible), or by setting a structure field (which is non-opaque), or by get and set functions (making the bitmask a redundant addition to the attributes objects).

Stack size is defined as an optional attribute because the very notion of a stack is inherently machine-dependent. Some implementations may not be able to change the size of the stack, for example, and others may not need to because stack pages may be discontiguous and can be allocated and released on demand.

The attribute mechanism has been designed in large measure for extensibility. Future extensions to the attribute mechanism or to any attributes object defined in this volume of IEEE Std 1003.1-200x has to be done with care so as not to affect binary-compatibility.

Attributes objects, even if allocated by means of dynamic allocation functions such as *malloc*( ), may have their size fixed at compile time. This means, for example, a *pthread_create*( ) in an implementation with extensions to the **pthread_attr_t** cannot look beyond the area that the binary application assumes is valid. This suggests that implementations should maintain a size field in the attributes object, as well as possibly version information, if extensions in different

directions (possibly by different vendors) are to be accommodated.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_getstackaddr( ), *pthread_attr_getstacksize*( ), *pthread_attr_getdetachstate*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_attr_destroy*( ) and *pthread_attr_init*( ) functions marked as part of the Threads option.

IEEE PASC Interpretation 1003.1 #107 is applied, noting that the effect of initializing an already initialized thread attributes object is undefined.

31437 **NAME**

31438 pthread_attr_getdetachstate, pthread_attr_setdetachstate — get and set detachstate attribute

31439 **SYNOPSIS**

31440 THR
```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
    int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```
31444

31445 **DESCRIPTION**

31446 The *detachstate* attribute controls whether the thread is created in a detached state. If the thread
31447 is created detached, then use of the ID of the newly created thread by the *pthread_detach*( ) or
31448 *pthread_join*( ) function is an error.

31449 The *pthread_attr_getdetachstate*( ) and *pthread_attr_setdetachstate*( ) functions, respectively, shall
31450 get and set the *detachstate* attribute in the *attr* object.

31451 For *pthread_attr_getdetachstate*( ), *detachstate* shall be set to either
31452 PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

31453 For *pthread_attr_setdetachstate*( ), the application shall set *detachstate* to either
31454 PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

31455 A value of PTHREAD_CREATE_DETACHED shall cause all threads created with *attr* to be in
31456 the detached state, whereas using a value of PTHREAD_CREATE_JOINABLE shall cause all
31457 threads created with *attr* to be in the joinable state. The default value of the *detachstate* attribute
31458 shall be PTHREAD_CREATE_JOINABLE.

31459 **RETURN VALUE**

31460 Upon successful completion, *pthread_attr_getdetachstate*( ) and *pthread_attr_setdetachstate*( ) shall
31461 return a value of 0; otherwise, an error number shall be returned to indicate the error.

31462 The *pthread_attr_getdetachstate*( ) function stores the value of the *detachstate* attribute in *detachstate*
31463 if successful.

31464 **ERRORS**

31465 The *pthread_attr_setdetachstate*( ) function shall fail if:

31466 [EINVAL] The value of *detachstate* was not valid

31467 These functions shall not return an error code of [EINTR].

31468 **EXAMPLES**

31469 None.

31470 **APPLICATION USAGE**

31471 None.

31472 **RATIONALE**

31473 None.

31474 **FUTURE DIRECTIONS**

31475 None.

31476 **SEE ALSO**

31477 *pthread_attr_destroy*( ), *pthread_attr_getstackaddr*( ), *pthread_attr_getstacksize*( ), *pthread_create*( ), the
31478 Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

31479 **CHANGE HISTORY**

31480 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31481 **Issue 6**

31482 The *pthread_attr_setdetachstate*( ) and *pthread_attr_getdetachstate*( ) functions are marked as part of
31483 the Threads option.

31484 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

31485 **NAME**

31486 pthread_attr_getguardsize, pthread_attr_setguardsize — get and set the thread guardsize
31487 attribute

31488 **SYNOPSIS**

31489 XSI
```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
    size_t *restrict guardsize);
int pthread_attr_setguardsize(pthread_attr_t *attr,
    size_t guardsize);
```
31494

31495 **DESCRIPTION**

31496 The *pthread_attr_getguardsize*( ) function shall get the *guardsize* attribute in the *attr* object. This |
31497 attribute shall be returned in the *guardsize* parameter. |

31498 The *pthread_attr_setguardsize*( ) function shall set the *guardsize* attribute in the *attr* object. The new |
31499 value of this attribute shall be obtained from the *guardsize* parameter. If *guardsize* is zero, a guard |
31500 area shall not be provided for threads created with *attr*. If *guardsize* is greater than zero, a guard
31501 area of at least size *guardsize* bytes shall be provided for each thread created with *attr*. |

31502 The *guardsize* attribute controls the size of the guard area for the created thread's stack. The |
31503 *guardsize* attribute provides protection against overflow of the stack pointer. If a thread's stack is |
31504 created with guard protection, the implementation allocates extra memory at the overflow end |
31505 of the stack as a buffer against stack overflow of the stack pointer. If an application overflows |
31506 into this buffer an error shall result (possibly in a SIGSEGV signal being delivered to the thread). |

31507 A conforming implementation may round up the value contained in *guardsize* to a multiple of |
31508 the configurable system variable {PAGESIZE} (see **<sys/mman.h>**). If an implementation
31509 rounds up the value of *guardsize* to a multiple of {PAGESIZE}, a call to *pthread_attr_getguardsize*( )
31510 specifying *attr* shall store in the *guardsize* parameter the guard size specified by the previous
31511 *pthread_attr_setguardsize*( ) function call.

31512 The default value of the *guardsize* attribute is {PAGESIZE} bytes. The actual value of {PAGESIZE} |
31513 is implementation-defined. |

31514 If the *stackaddr* or *stack* attribute has been set (that is, the caller is allocating and managing its
31515 own thread stacks), the *guardsize* attribute shall be ignored and no protection shall be provided |
31516 by the implementation. It is the responsibility of the application to manage stack overflow along |
31517 with stack allocation and management in this case.

31518 **RETURN VALUE**

31519 If successful, the *pthread_attr_getguardsize*( ) and *pthread_attr_setguardsize*( ) functions shall return
31520 zero; otherwise, an error number shall be returned to indicate the error.

31521 **ERRORS**

31522 The *pthread_attr_getguardsize*( ) and *pthread_attr_setguardsize*( ) functions shall fail if:

31523 [EINVAL] The attribute *attr* is invalid.

31524 [EINVAL] The parameter *guardsize* is invalid.

31525 These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *guardsize* attribute is provided to the application for two reasons:

1. Overflow protection can potentially result in wasted system resources. An application that creates a large number of threads, and which knows its threads never overflow their stack, can save system resources by turning off guard areas.
2. When threads allocate large data structures on the stack, large guard areas may be needed to detect stack overflow.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

In the ERRORS section, a third [EINVAL] error condition is removed as it is covered by the second error condition.

The **restrict** keyword is added to the *pthread_attr_getguardsize*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

31548 **NAME**

31549 pthread_attr_getinheritsched, pthread_attr_setinheritsched — get and set inheritsched attribute
31550 (**REALTIME THREADS**)

31551 **SYNOPSIS**

31552 THR TPS
```
#include <pthread.h>

int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr,
    int *restrict inheritsched);
int pthread_attr_setinheritsched(pthread_attr_t *attr,
    int inheritsched);
```
31557

31558 **DESCRIPTION**

31559 The *pthread_attr_getinheritsched*( ), and *pthread_attr_setinheritsched*( ) functions, respectively, shall
31560 get and set the *inheritsched* attribute in the *attr* argument.

31561 When the attributes objects are used by *pthread_create*( ), the *inheritsched* attribute determines
31562 how the other scheduling attributes of the created thread shall be set. |

31563 PTHREAD_INHERIT_SCHED
31564 Specifies that the thread scheduling attributes shall be inherited from the creating thread,
31565 and the scheduling attributes in this *attr* argument shall be ignored.

31566 PTHREAD_EXPLICIT_SCHED
31567 Specifies that the thread scheduling attributes shall be set to the corresponding values from
31568 this attributes object.

31569 The symbols PTHREAD_INHERIT_SCHED and PTHREAD_EXPLICIT_SCHED are defined in |
31570 the **<pthread.h>** header. |

31571 The following ''thread scheduling attributes'' defined by IEEE Std 1003.1-200x are affected by |
31572 the *inheritsched* attribute: scheduling policy (*schedpolicy*), scheduling parameters (*schedparam*), |
31573 and scheduling contention scope (*contentionscope*). |

31574 **RETURN VALUE**

31575 If successful, the *pthread_attr_getinheritsched*( ) and *pthread_attr_setinheritsched*( ) functions shall
31576 return zero; otherwise, an error number shall be returned to indicate the error.

31577 **ERRORS**

31578 The *pthread_attr_setinheritsched*( ) function may fail if:

31579 [EINVAL] The value of *inheritsched* is not valid.

31580 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

31581 These functions shall not return an error code of [EINTR].

31582 **EXAMPLES**

31583 None.

31584 **APPLICATION USAGE**

31585 After these attributes have been set, a thread can be created with the specified attributes using
31586 *pthread_create*( ). Using these routines does not affect the current running thread.

31587 **RATIONALE**

31588 None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_destroy( ), *pthread_attr_getscope*( ), *pthread_attr_getschedpolicy*( ), *pthread_attr_getschedparam*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_attr_getinheritsched*( ) and *pthread_attr_setinheritsched*( ) functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The **restrict** keyword is added to the *pthread_attr_getinheritsched*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

31605 **NAME**

31606 pthread_attr_getschedparam, pthread_attr_setschedparam — get and set schedparam attribute

31607 **SYNOPSIS**

31608 THR
```
#include <pthread.h>

int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
    struct sched_param *restrict param);
int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
    const struct sched_param *restrict param);
```

31613

31614 **DESCRIPTION**

31615 The *pthread_attr_getschedparam*( ), and *pthread_attr_setschedparam*( ) functions, respectively, shall
31616 get and set the scheduling parameter attributes in the *attr* argument. The contents of the *param*
31617 structure are defined in the **<sched.h>** header. For the SCHED_FIFO and SCHED_RR policies,
31618 the only required member of *param* is *sched_priority*.

31619 TSP For the SCHED_SPORADIC policy, the required members of the *param* structure are
31620 *sched_priority*, *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, and
31621 *sched_ss_max_repl*. The specified *sched_ss_repl_period* must be greater than or equal to the
31622 specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.
31623 The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the
31624 function to succeed; if not, the function shall fail.

31625 **RETURN VALUE**

31626 If successful, the *pthread_attr_getschedparam*( ) and *pthread_attr_setschedparam*( ) functions shall
31627 return zero; otherwise, an error number shall be returned to indicate the error.

31628 **ERRORS**

31629 The *pthread_attr_setschedparam*( ) function may fail if:

31630 [EINVAL] The value of *param* is not valid.

31631 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

31632 These functions shall not return an error code of [EINTR].

31633 **EXAMPLES**

31634 None.

31635 **APPLICATION USAGE**

31636 After these attributes have been set, a thread can be created with the specified attributes using
31637 *pthread_create*( ). Using these routines does not affect the current running thread.

31638 **RATIONALE**

31639 None.

31640 **FUTURE DIRECTIONS**

31641 None.

31642 **SEE ALSO**

31643 *pthread_attr_destroy*( ), *pthread_attr_getscope*( ), *pthread_attr_getinheritsched*( ),
31644 *pthread_attr_getschedpolicy*( ), *pthread_create*( ), the Base Definitions volume of
31645 IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>**

31646 **CHANGE HISTORY**

31647 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31648 **Issue 6**

31649 The *pthread_attr_getschedparam*( ) and *pthread_attr_setschedparam*( ) functions are marked as part
31650 of the Threads option.

31651 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

31652 The **restrict** keyword is added to the *pthread_attr_getschedparam*( ) and
31653 *pthread_attr_setschedparam*( ) prototypes for alignment with the ISO / IEC 9899: 1999 standard.

31654 **NAME**

31655 pthread_attr_getschedpolicy, pthread_attr_setschedpolicy — get and set schedpolicy attribute
31656 (**REALTIME THREADS**)

31657 **SYNOPSIS**

31658 THR TPS
```
#include <pthread.h>

int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
    int *restrict policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```
31662

31663 **DESCRIPTION**

31664 The *pthread_attr_getschedpolicy*( ) and *pthread_attr_setschedpolicy*( ) functions, respectively, shall
31665 get and set the *schedpolicy* attribute in the *attr* argument.

31666 The supported values of *policy* shall include SCHED_FIFO, SCHED_RR, and SCHED_OTHER, |
31667 which are defined in the **<sched.h>** header. When threads executing with the scheduling policy |
31668 TSP SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC are waiting on a mutex, they shall acquire |
31669 the mutex in priority order when the mutex is unlocked. |

31670 **RETURN VALUE**

31671 If successful, the *pthread_attr_getschedpolicy*( ) and *pthread_attr_setschedpolicy*( ) functions shall
31672 return zero; otherwise, an error number shall be returned to indicate the error.

31673 **ERRORS**

31674 The *pthread_attr_setschedpolicy*( ) function may fail if:

31675 [EINVAL] The value of *policy* is not valid.

31676 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

31677 These functions shall not return an error code of [EINTR].

31678 **EXAMPLES**

31679 None.

31680 **APPLICATION USAGE**

31681 After these attributes have been set, a thread can be created with the specified attributes using
31682 *pthread_create*( ). Using these routines does not affect the current running thread.

31683 **RATIONALE**

31684 None.

31685 **FUTURE DIRECTIONS**

31686 None.

31687 **SEE ALSO**

31688 *pthread_attr_destroy*( ), *pthread_attr_getscope*( ), *pthread_attr_getinheritsched*( ),
31689 *pthread_attr_getschedparam*( ), *pthread_create*( ), the Base Definitions volume of
31690 IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>**

31691 **CHANGE HISTORY**

31692 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31693 Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_attr_getschedpolicy*( ) and *pthread_attr_setschedpolicy*( ) functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *pthread_attr_getschedpolicy*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

31702 **NAME**

31703 pthread_attr_getscope, pthread_attr_setscope — get and set contentionscope attribute
31704 (**REALTIME THREADS**)

31705 **SYNOPSIS**

31706 THR TPS
```
#include <pthread.h>

int pthread_attr_getscope(const pthread_attr_t *restrict attr,
    int *restrict contentionscope);
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```
31710

31711 **DESCRIPTION**

31712 The *pthread_attr_getscope*( ) and *pthread_attr_setscope*( ) functions, respectively, shall get and set |
31713 the *contentionscope* attribute in the *attr* object. |

31714 The *contentionscope* attribute may have the values PTHREAD_SCOPE_SYSTEM, signifying
31715 system scheduling contention scope, or PTHREAD_SCOPE_PROCESS, signifying process
31716 scheduling contention scope. The symbols PTHREAD_SCOPE_SYSTEM and |
31717 PTHREAD_SCOPE_PROCESS are defined in the **<pthread.h>** header. |

31718 **RETURN VALUE**

31719 If successful, the *pthread_attr_getscope*( ) and *pthread_attr_setscope*( ) functions shall return zero;
31720 otherwise, an error number shall be returned to indicate the error.

31721 **ERRORS**

31722 The *pthread_attr_setscope*( ) function may fail if:

31723 [EINVAL] The value of *contentionscope* is not valid.

31724 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

31725 These functions shall not return an error code of [EINTR].

31726 **EXAMPLES**

31727 None.

31728 **APPLICATION USAGE**

31729 After these attributes have been set, a thread can be created with the specified attributes using
31730 *pthread_create*( ). Using these routines does not affect the current running thread.

31731 **RATIONALE**

31732 None.

31733 **FUTURE DIRECTIONS**

31734 None.

31735 **SEE ALSO**

31736 *pthread_attr_destroy*( ), *pthread_attr_getinheritsched*( ), *pthread_attr_getschedpolicy*( ),
31737 *pthread_attr_getschedparam*( ), *pthread_create*( ), the Base Definitions volume of
31738 IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>**

31739 **CHANGE HISTORY**

31740 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31741 Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_attr_getscope*( ) and *pthread_attr_setscope*( ) functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The **restrict** keyword is added to the *pthread_attr_getscope*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

31749 **NAME**

31750 pthread_attr_getstack, pthread_attr_setstack — get and set stack attributes

31751 **SYNOPSIS**

31752 THR
```
#include <pthread.h>
```

31753 TSA TSS
```
int pthread_attr_getstack(const pthread_attr_t *restrict attr,
    void **restrict stackaddr, size_t *restrict stacksize);
int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
    size_t stacksize);
```

31757

31758 **DESCRIPTION**

31759 The *pthread_attr_getstack*( ) and *pthread_attr_setstack*( ) functions, respectively, shall get and set
31760 the thread creation stack attributes *stackaddr* and *stacksize* in the *attr* object.

31761 The stack attributes specify the area of storage to be used for the created thread's stack. The base
31762 (lowest addressable byte) of the storage shall be *stackaddr*, and the size of the storage shall be
31763 *stacksize* bytes. The *stacksize* shall be at least {PTHREAD_STACK_MIN}. The *stackaddr* shall be
31764 aligned appropriately to be used as a stack; for example, *pthread_attr_setstack*( ) may fail with
31765 [EINVAL] if (*stackaddr* & 0x7) is not 0. All pages within the stack described by *stackaddr* and
31766 *stacksize* shall be both readable and writable by the thread.

31767 **RETURN VALUE**

31768 Upon successful completion, these functions shall return a value of 0; otherwise, an error
31769 number shall be returned to indicate the error.

31770 The *pthread_attr_getstack*( ) function shall store the stack attribute values in *stackaddr* and *stacksize*
31771 if successful.

31772 **ERRORS**

31773 The *pthread_attr_setstack*( ) function shall fail if:

31774 [EINVAL] The value of *stacksize* is less than {PTHREAD_STACK_MIN} or exceeds an
31775 implementation-defined limit.

31776 The *pthread_attr_setstack*( ) function may fail if:

31777 [EINVAL] The value of *stackaddr* does not have proper alignment to be used as a stack, or
31778 if (*stackaddr* + *stacksize*) lacks proper alignment.

31779 [EACCES] The stack page(s) described by *stackaddr* and *stacksize* are not both readable
31780 and writable by the thread.

31781 These functions shall not return an error code of [EINTR].

31782 **EXAMPLES**

31783 None.

31784 **APPLICATION USAGE**

31785 These functions are appropriate for use by applications in an environment where the stack for a
31786 thread must be placed in some particular region of memory.

31787 While it might seem that an application could detect stack overflow by providing a protected
31788 page outside the specified stack region, this cannot be done portably. Implementations are free
31789 to place the thread's initial stack pointer anywhere within the specified region to accommodate
31790 the machine's stack pointer behavior and allocation requirements. Furthermore, on some
31791 architectures, such as the IA-64, ''overflow'' might mean that two separate stack pointers
31792 allocated within the region will overlap somewhere in the middle of the region.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_init( ), *pthread_attr_setdetachstate*( ), *pthread_attr_setstacksize*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Developed as an XSI extension and brought into the BASE by IEEE PASC Interpretation 1003.1 #101.

31803 **NAME**

31804 pthread_attr_getstackaddr, pthread_attr_setstackaddr — get and set stackaddr attribute

31805 **SYNOPSIS**

31806 THR TSA
```
#include <pthread.h>
```

31807 OB
```
int pthread_attr_getstackaddr(const pthread_attr_t *restrict attr,
    void **restrict stackaddr);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

31810

31811 **DESCRIPTION**

31812 The *pthread_attr_getstackaddr*( ) and *pthread_attr_setstackaddr*( ) functions, respectively, shall get
31813 and set the thread creation *stackaddr* attribute in the *attr* object.

31814 The *stackaddr* attribute specifies the location of storage to be used for the created thread's stack.
31815 The size of the storage shall be at least {PTHREAD_STACK_MIN}.

31816 **RETURN VALUE**

31817 Upon successful completion, *pthread_attr_getstackaddr*( ) and *pthread_attr_setstackaddr*( ) shall
31818 return a value of 0; otherwise, an error number shall be returned to indicate the error.

31819 The *pthread_attr_getstackaddr*( ) function stores the *stackaddr* attribute value in *stackaddr* if
31820 successful.

31821 **ERRORS**

31822 No errors are defined.

31823 These functions shall not return an error code of [EINTR].

31824 **EXAMPLES**

31825 None.

31826 **APPLICATION USAGE**

31827 The specification of the *stackaddr* attribute presents several ambiguities that make portable use of
31828 these interfaces impossible. The description of the single address parameter as a ''stack'' does
31829 not specify a particular relationship between the address and the ''stack'' implied by that
31830 address. For example, the address may be taken as the low memory address of a buffer intended
31831 for use as a stack, or it may be taken as the address to be used as the initial stack pointer register
31832 value for the new thread. These two are not the same except for a machine on which the stack
31833 grows ''up'' from low memory to high, and on which a ''push'' operation first stores the value in
31834 memory and then increments the stack pointer register. Further, on a machine where the stack
31835 grows ''down'' from high memory to low, interpretation of the address as the ''low memory''
31836 address requires a determination of the intended size of the stack. IEEE Std 1003.1-200x has
31837 introduced the new interfaces *pthread_attr_setstack*( ) and *pthread_attr_getstack*( ) to resolve these
31838 ambiguities.

31839 **RATIONALE**

31840 None.

31841 **FUTURE DIRECTIONS**

31842 None.

31843 **SEE ALSO**

31844 *pthread_attr_destroy*( ), *pthread_attr_getdetachstate*( ), *pthread_attr_getstack*( ),
31845 *pthread_attr_getstacksize*( ), *pthread_attr_setstack*( ), *pthread_create*( ), the Base Definitions volume
31846 of IEEE Std 1003.1-200x, **<limits.h>**, **<pthread.h>**

31847 **CHANGE HISTORY**

31848 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31849 **Issue 6**

31850 The *pthread_attr_getstackaddr*( ) and *pthread_attr_setstackaddr*( ) functions are marked as part of
31851 the Threads and Thread Stack Address Attribute options.

31852 The **restrict** keyword is added to the *pthread_attr_getstackaddr*( ) prototype for alignment with the
31853 ISO / IEC 9899: 1999 standard.

31854 These functions are marked obsolescent.

31855 **NAME**

31856 pthread_attr_getstacksize, pthread_attr_setstacksize — get and set stacksize attribute

31857 **SYNOPSIS**

31858 THR TSA
```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
    size_t *restrict stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```
31862

31863 **DESCRIPTION**

31864 The *pthread_attr_getstacksize*( ) and *pthread_attr_setstacksize*( ) functions, respectively, shall get
31865 and set the thread creation *stacksize* attribute in the *attr* object.

31866 The *stacksize* attribute shall define the minimum stack size (in bytes) allocated for the created
31867 threads stack.

31868 **RETURN VALUE**

31869 Upon successful completion, *pthread_attr_getstacksize*( ) and *pthread_attr_setstacksize*( ) shall
31870 return a value of 0; otherwise, an error number shall be returned to indicate the error.

31871 The *pthread_attr_getstacksize*( ) function stores the *stacksize* attribute value in *stacksize* if
31872 successful.

31873 **ERRORS**

31874 The *pthread_attr_setstacksize*( ) function shall fail if:

31875 [EINVAL] The value of *stacksize* is less than {PTHREAD_STACK_MIN} or exceeds a
31876 system-imposed limit.

31877 These functions shall not return an error code of [EINTR].

31878 **EXAMPLES**

31879 None.

31880 **APPLICATION USAGE**

31881 None.

31882 **RATIONALE**

31883 None.

31884 **FUTURE DIRECTIONS**

31885 None.

31886 **SEE ALSO**

31887 *pthread_attr_destroy*( ), *pthread_attr_getstackaddr*( ), *pthread_attr_getdetachstate*( ), *pthread_create*( ),
31888 the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<pthread.h>**

31889 **CHANGE HISTORY**

31890 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31891 **Issue 6**

31892 The *pthread_attr_getstacksize*( ) and *pthread_attr_setstacksize*( ) functions are marked as part of the
31893 Threads and Thread Stack Address Attribute options.

31894 The **restrict** keyword is added to the *pthread_attr_getstacksize*( ) prototype for alignment with the
31895 ISO / IEC 9899: 1999 standard.

31896 **NAME**

31897 pthread_attr_init — initialize threads attributes object

31898 **SYNOPSIS**

31899 THR
```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
```
31900

31901

31902 **DESCRIPTION**

31903 Refer to *pthread_attr_destroy*( ).

31904 **NAME**

31905 pthread_attr_setdetachstate — set detachstate attribute

31906 **SYNOPSIS**

31907 THR
```
#include <pthread.h>
```

31908
```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

31909

31910 **DESCRIPTION**

31911 Refer to *pthread_attr_getdetachstate*( ).

31912 **NAME**

31913 pthread_attr_setguardsize — set thread guardsize attribute

31914 **SYNOPSIS**

31915 XSI
```
#include <pthread.h>

int pthread_attr_setguardsize(pthread_attr_t *attr,
    size_t guardsize);
```
31916
31917
31918

31919 **DESCRIPTION**

31920 Refer to *pthread_attr_getguardsize*( ).

31921 **NAME**

31922 pthread_attr_setinheritsched — set inheritsched attribute (**REALTIME THREADS**)

31923 **SYNOPSIS**

31924 THR TPS
```
#include <pthread.h>

int pthread_attr_setinheritsched(pthread_attr_t *attr,
    int inheritsched);
```
31927

31928 **DESCRIPTION**

31929 Refer to *pthread_attr_getinheritsched*( ).

31930 **NAME**

31931 pthread_attr_setschedparam — set schedparam attribute

31932 **SYNOPSIS**

31933 THR
```
#include <pthread.h>

int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
    const struct sched_param *restrict param);
```
31934

31935

31936

31937 **DESCRIPTION**

31938 Refer to *pthread_attr_getschedparam*( ).

31939 **NAME**

31940 pthread_attr_setschedpolicy — set schedpolicy attribute (**REALTIME THREADS**)

31941 **SYNOPSIS**

31942 THR TPS
```
#include <pthread.h>

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```
31943

31944

31945 **DESCRIPTION**

31946 Refer to *pthread_attr_getschedpolicy*( ).

31947 **NAME**

31948 pthread_attr_setscope — set contentionscope attribute (**REALTIME THREADS**)

31949 **SYNOPSIS**

31950 THR TPS
```
#include <pthread.h>
```

31951
```
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

31952

31953 **DESCRIPTION**

31954 Refer to *pthread_attr_getscope*( ).

31955 **NAME**

31956 pthread_attr_setstack — set stack attribute

31957 **SYNOPSIS**

31958 XSI
```
#include <pthread.h>

int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
    size_t stacksize);
```
31959
31960
31961

31962 **DESCRIPTION**

31963 Refer to *pthread_attr_getstack*( ).

NAME

pthread_attr_setstackaddr — set stackaddr attribute

SYNOPSIS

THR TSA
```
#include <pthread.h>
```

OB
```
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

DESCRIPTION

Refer to *pthread_attr_getstackaddr*( ).

31972 **NAME**

31973 pthread_attr_setstacksize — set stacksize attribute

31974 **SYNOPSIS**

31975 THR TSA
```
#include <pthread.h>

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```
31976

31977

31978 **DESCRIPTION**

31979 Refer to *pthread_attr_getstacksize*( ).

31980 **NAME**

31981 pthread_barrier_destroy, pthread_barrier_init — destroy and initialize a barrier object
31982 (**ADVANCED REALTIME THREADS**)

31983 **SYNOPSIS**

31984 THR BAR
```
#include <pthread.h>

int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
    const pthread_barrierattr_t *restrict attr, unsigned count);
```
31988

31989 **DESCRIPTION**

31990 The *pthread_barrier_destroy*( ) function shall destroy the barrier referenced by *barrier* and release
31991 any resources used by the barrier. The effect of subsequent use of the barrier is undefined until
31992 the barrier is reinitialized by another call to *pthread_barrier_init*( ). An implementation may use
31993 this function to set *barrier* to an invalid value. The results are undefined if
31994 *pthread_barrier_destroy*( ) is called when any thread is blocked on the barrier, or if this function is
31995 called with an uninitialized barrier.

31996 The *pthread_barrier_init*( ) function shall allocate any resources required to use the barrier
31997 referenced by *barrier* and shall initialize the barrier with attributes referenced by *attr*. If *attr* is
31998 NULL, the default barrier attributes shall be used; the effect is the same as passing the address of
31999 a default barrier attributes object. The results are undefined if *pthread_barrier_init*( ) is called
32000 when any thread is blocked on the barrier (that is, has not returned from the
32001 *pthread_barrier_wait*( ) call). The results are undefined if a barrier is used without first being
32002 initialized. The results are undefined if *pthread_barrier_init*( ) is called specifying an already
32003 initialized barrier.

32004 The *count* argument specifies the number of threads that must call *pthread_barrier_wait*( ) before
32005 any of them successfully return from the call. The value specified by *count* must be greater than
32006 zero.

32007 If the *pthread_barrier_init*( ) function fails, the barrier shall not be initialized and the contents of |
32008 *barrier* are undefined.

32009 Only the object referenced by *barrier* may be used for performing synchronization. The result of
32010 referring to copies of that object in calls to *pthread_barrier_destroy*( ) or *pthread_barrier_wait*( ) is
32011 undefined.

32012 **RETURN VALUE**

32013 Upon successful completion, these functions shall return zero; otherwise, an error number shall
32014 be returned to indicate the error.

32015 **ERRORS**

32016 The *pthread_barrier_destroy*( ) function may fail if:

32017 [EBUSY] The implementation has detected an attempt to destroy a barrier while it is in
32018 use (for example, while being used in a *pthread_barrier_wait*( ) call) by another
32019 thread.

32020 [EINVAL] The value specified by *barrier* is invalid.

32021 The *pthread_barrier_init*( ) function shall fail if:

32022 [EAGAIN] The system lacks the necessary resources to initialize another barrier.

32023 [EINVAL] The value specified by *count* is equal to zero.

[ENOMEM] Insufficient memory exists to initialize the barrier.

The *pthread_barrier_init*( ) function may fail if:

[EBUSY] The implementation has detected an attempt to reinitialize a barrier while it is in use (for example, while being used in a *pthread_barrier_wait*( ) call) by another thread.

[EINVAL] The value specified by *attr* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_barrier_destroy*( ) and *pthread_barrier_init*( ) functions are part of the Barriers option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_barrier_wait( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

32044 **NAME**

32045 pthread_barrier_init — initialize a barrier object (**ADVANCED REALTIME THREADS**)

32046 **SYNOPSIS**

32047 THR BAR
```
#include <pthread.h>

int pthread_barrier_init(pthread_barrier_t *restrict barrier,
    const pthread_barrierattr_t *restrict attr, unsigned count);
```
32048

32049

32050

32051 **DESCRIPTION**

32052 Refer to *pthread_barrier_destroy*( ).

32053 **NAME**

32054 pthread_barrier_wait — synchronize at a barrier (**ADVANCED REALTIME THREADS**)

32055 **SYNOPSIS**

32056 THR BAR
```
#include <pthread.h>

int pthread_barrier_wait(pthread_barrier_t *barrier);
```
32058

32059 **DESCRIPTION**

32060 The *pthread_barrier_wait*( ) function shall synchronize participating threads at the barrier
32061 referenced by *barrier*. The calling thread shall block until the required number of threads have |
32062 called *pthread_barrier_wait*( ) specifying the barrier. |

32063 When the required number of threads have called *pthread_barrier_wait*( ) specifying the barrier,
32064 the constant PTHREAD_BARRIER_SERIAL_THREAD shall be returned to one unspecified
32065 thread and zero shall be returned to each of the remaining threads. At this point, the barrier shall
32066 be reset to the state it had as a result of the most recent *pthread_barrier_init*( ) function that
32067 referenced it.

32068 The constant PTHREAD_BARRIER_SERIAL_THREAD is defined in **<pthread.h>** and its value |
32069 shall be distinct from any other value returned by *pthread_barrier_wait*( ).

32070 The results are undefined if this function is called with an uninitialized barrier.

32071 If a signal is delivered to a thread blocked on a barrier, upon return from the signal handler the
32072 thread shall resume waiting at the barrier if the barrier wait has not completed (that is, if the
32073 required number of threads have not arrived at the barrier during the execution of the signal
32074 handler); otherwise, the thread shall continue as normal from the completed barrier wait. Until
32075 the thread in the signal handler returns from it, it is unspecified whether other threads may
32076 proceed past the barrier once they have all reached it.

32077 A thread that has blocked on a barrier shall not prevent any unblocked thread that is eligible to
32078 use the same processing resources from eventually making forward progress in its execution.
32079 Eligibility for processing resources shall be determined by the scheduling policy.

32080 **RETURN VALUE**

32081 Upon successful completion, the *pthread_barrier_wait*( ) function shall return
32082 PTHREAD_BARRIER_SERIAL_THREAD for a single (arbitrary) thread synchronized at the
32083 barrier and zero for each of the other threads. Otherwise, an error number shall be returned to
32084 indicate the error.

32085 **ERRORS**

32086 The *pthread_barrier_wait*( ) function may fail if:

32087 [EINVAL] The value specified by *barrier* does not refer to an initialized barrier object.

32088 This function shall not return an error code of [EINTR].

32089 **EXAMPLES**

32090 None.

32091 **APPLICATION USAGE**

32092 Applications using this function may be subject to priority inversion, as discussed in the Base
32093 Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

32094 The *pthread_barrier_wait*( ) function is part of the Barriers option and need not be provided on all
32095 implementations.

32096 **RATIONALE**

32097 None.

32098 **FUTURE DIRECTIONS**

32099 None.

32100 **SEE ALSO**

32101 *pthread_barrier_destroy*( ), *pthread_barrier_init*( ), the Base Definitions volume of
32102 IEEE Std 1003.1-200x, **<pthread.h>**

32103 **CHANGE HISTORY**

32104 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

32105 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

32106 **NAME**

32107 pthread_barrierattr_destroy, pthread_barrierattr_init — destroy and initialize barrier attributes
32108 object (**ADVANCED REALTIME THREADS**)

32109 **SYNOPSIS**

32110 THR BAR
```
#include <pthread.h>

int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```
32113

32114 **DESCRIPTION**

32115 The *pthread_barrierattr_destroy*( ) function shall destroy a barrier attributes object. A destroyed |
32116 *attr* attributes object can be reinitialized using *pthread_barrierattr_init*( ); the results of otherwise |
32117 referencing the object after it has been destroyed are undefined. An implementation may cause |
32118 *pthread_barrierattr_destroy*( ) to set the object referenced by *attr* to an invalid value.

32119 The *pthread_barrierattr_init*( ) function shall initialize a barrier attributes object *attr* with the
32120 default value for all of the attributes defined by the implementation.

32121 Results are undefined if *pthread_barrierattr_init*( ) is called specifying an already initialized *attr* |
32122 attributes object. |

32123 After a barrier attributes object has been used to initialize one or more barriers, any function
32124 affecting the attributes object (including destruction) shall not affect any previously initialized |
32125 barrier. |

32126 **RETURN VALUE**

32127 If successful, the *pthread_barrierattr_destroy*( ) and *pthread_barrierattr_init*( ) functions shall return
32128 zero; otherwise, an error number shall be returned to indicate the error.

32129 **ERRORS**

32130 The *pthread_barrierattr_destroy*( ) function may fail if:

32131 [EINVAL] The value specified by *attr* is invalid.

32132 The *pthread_barrierattr_init*( ) function shall fail if:

32133 [ENOMEM] Insufficient memory exists to initialize the barrier attributes object.

32134 These functions shall not return an error code of [EINTR].

32135 **EXAMPLES**

32136 None.

32137 **APPLICATION USAGE**

32138 The *pthread_barrierattr_destroy*( ) and *pthread_barrierattr_init*( ) functions are part of the Barriers
32139 option and need not be provided on all implementations.

32140 **RATIONALE**

32141 None.

32142 **FUTURE DIRECTIONS**

32143 None.

32144 **SEE ALSO**

32145 *pthread_barrierattr_getpshared*( ), *pthread_barrierattr_setpshared*( ), the Base Definitions volume of
32146 IEEE Std 1003.1-200x, **<pthread.h>**.

32147 **CHANGE HISTORY**

32148 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

32149 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

32150 **NAME**

32151 pthread_barrierattr_getpshared, pthread_barrierattr_setpshared — get and set process-shared
32152 attribute of barrier attributes object (**ADVANCED REALTIME THREADS**)

32153 **SYNOPSIS**

32154 THR
```
#include <pthread.h>
```

32155 BAR TSH
```
int pthread_barrierattr_getpshared(
    const pthread_barrierattr_t *restrict attr,
    int *restrict pshared);
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
    int pshared);
```

32160

32161 **DESCRIPTION**

32162 The *pthread_barrierattr_getpshared*( ) function shall obtain the value of the process-shared |
32163 attribute from the attributes object referenced by *attr*. The *pthread_barrierattr_setpshared*( ) |
32164 function shall set the process-shared attribute in an initialized attributes object referenced by |
32165 *attr*. |

32166 The process-shared attribute is set to PTHREAD_PROCESS_SHARED to permit a barrier to be |
32167 operated upon by any thread that has access to the memory where the barrier is allocated. If the
32168 process-shared attribute is PTHREAD_PROCESS_PRIVATE, the barrier shall only be operated
32169 upon by threads created within the same process as the thread that initialized the barrier; if
32170 threads of different processes attempt to operate on such a barrier, the behavior is undefined.
32171 The default value of the attribute shall be PTHREAD_PROCESS_PRIVATE. Both constants |
32172 PTHREAD_PROCESS_SHARED and PTHREAD_PROCESS_PRIVATE are defined in |
32173 **<pthread.h>**.

32174 Additional attributes, their default values, and the names of the associated functions to get and |
32175 set those attribute values are implementation-defined.

32176 **RETURN VALUE**

32177 If successful, the *pthread_barrierattr_getpshared*( ) function shall return zero and store the value of
32178 the process-shared attribute of *attr* into the object referenced by the *pshared* parameter.
32179 Otherwise, an error number shall be returned to indicate the error.

32180 If successful, the *pthread_barrierattr_setpshared*( ) function shall return zero; otherwise, an error
32181 number shall be returned to indicate the error.

32182 **ERRORS**

32183 These functions may fail if:

32184 [EINVAL] The value specified by *attr* is invalid.

32185 The *pthread_barrierattr_setpshared*( ) function may fail if:

32186 [EINVAL] The new value specified for the process-shared attribute is not one of the legal
32187 values PTHREAD_PROCESS_SHARED or PTHREAD_PROCESS_PRIVATE.

32188 These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_barrierattr_getpshared*( ) and *pthread_barrierattr_setpshared*( ) functions are part of the Barriers option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_barrier_init( ), *pthread_barrierattr_destroy*( ), *pthread_barrierattr_init*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000

32203 **NAME**

32204 pthread_barrierattr_init — initialize barrier attributes object (**ADVANCED REALTIME**
32205 **THREADS**)

32206 **SYNOPSIS**

32207 THR BAR
```
#include <pthread.h>
```
32208
```
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```
32209

32210 **DESCRIPTION**

32211 Refer to *pthread_barrierattr_destroy*( ).

32212 **NAME**

32213 pthread_barrierattr_setpshared — set process-shared attribute of barrier attributes object
32214 (**ADVANCED REALTIME THREADS**)

32215 **SYNOPSIS**

32216 THR
```
#include <pthread.h>
```

32217 BAR TSH
```
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
    int pshared);
```

32218
32219

32220 **DESCRIPTION**

32221 Refer to *pthread_barrierattr_getpshared*( ).

32222 **NAME**

32223 pthread_cancel — cancel execution of a thread

32224 **SYNOPSIS**

32225 THR
```
#include <pthread.h>
```

32226
```
int pthread_cancel(pthread_t thread);
```

32227

32228 **DESCRIPTION**

32229 The *pthread_cancel*( ) function shall request that *thread* be canceled. The target thread's
32230 cancelability state and type determines when the cancelation takes effect. When the cancelation
32231 is acted on, the cancelation cleanup handlers for *thread* shall be called. When the last cancelation
32232 cleanup handler returns, the thread-specific data destructor functions shall be called for *thread*.
32233 When the last destructor function returns, *thread* shall be terminated.

32234 The cancelation processing in the target thread shall run asynchronously with respect to the
32235 calling thread returning from *pthread_cancel*( ).

32236 **RETURN VALUE**

32237 If successful, the *pthread_cancel*( ) function shall return zero; otherwise, an error number shall be
32238 returned to indicate the error.

32239 **ERRORS**

32240 The *pthread_cancel*( ) function may fail if:

32241 [ESRCH] No thread could be found corresponding to that specified by the given thread
32242 ID.

32243 The *pthread_cancel*( ) function shall not return an error code of [EINTR].

32244 **EXAMPLES**

32245 None.

32246 **APPLICATION USAGE**

32247 None.

32248 **RATIONALE**

32249 Two alternative functions were considered to sending the cancelation notification to a thread.
32250 One would be to define a new SIGCANCEL signal that had the cancelation semantics when
32251 delivered; the other was to define the new *pthread_cancel*( ) function, which would trigger the
32252 cancelation semantics.

32253 The advantage of a new signal was that so much of the delivery criteria were identical to that
32254 used when trying to deliver a signal that making cancelation notification a signal was seen as
32255 consistent. Indeed, many implementations implement cancelation using a special signal. On the
32256 other hand, there would be no signal functions that could be used with this signal except
32257 *pthread_kill*( ), and the behavior of the delivered cancelation signal would be unlike any
32258 previously existing defined signal.

32259 The benefits of a special function include the recognition that this signal would be defined
32260 because of the similar delivery criteria and that this is the only common behavior between a
32261 cancelation request and a signal. In addition, the cancelation delivery mechanism does not have
32262 to be implemented as a signal. There are also strong, if not stronger, parallels with language
32263 exception mechanisms than with signals that are potentially obscured if the delivery mechanism
32264 is visibly closer to signals.

32265 In the end, it was considered that as there were so many exceptions to the use of the new signal
32266 with existing signals functions that it would be misleading. A special function has resolved this

problem. This function was carefully defined so that an implementation wishing to provide the cancelation functions on top of signals could do so. The special function also means that implementations are not obliged to implement cancelation with signals.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_exit( ), *pthread_cond_wait*( ), *pthread_cond_timedwait*( ), *pthread_join*( ), *pthread_setcancelstate*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_cancel*( ) function is marked as part of the Threads option.

32279 **NAME**

32280 pthread_cleanup_pop, pthread_cleanup_push — establish cancelation handlers

32281 **SYNOPSIS**

32282 THR
```
#include <pthread.h>

void pthread_cleanup_pop(int execute);
void pthread_cleanup_push(void (*routine)(void*), void *arg);
```
32283
32284
32285

32286 **DESCRIPTION**

32287 The *pthread_cleanup_pop*( ) function shall remove the routine at the top of the calling thread's
32288 cancelation cleanup stack and optionally invoke it (if *execute* is non-zero).

32289 The *pthread_cleanup_push*( ) function shall push the specified cancelation cleanup handler *routine*
32290 onto the calling thread's cancelation cleanup stack. The cancelation cleanup handler shall be
32291 popped from the cancelation cleanup stack and invoked with the argument *arg* when:

32292 • The thread exits (that is, calls *pthread_exit*( )).

32293 • The thread acts upon a cancelation request.

32294 • The thread calls *pthread_cleanup_pop*( ) with a non-zero *execute* argument.

32295 These functions may be implemented as macros. The application shall ensure that they appear
32296 as statements, and in pairs within the same lexical scope (that is, the *pthread_cleanup_push*( )
32297 macro may be thought to expand to a token list whose first token is `'{'` with
32298 *pthread_cleanup_pop*( ) expanding to a token list whose last token is the corresponding `'}'`).

32299 The effect of calling *longjmp*( ) or *siglongjmp*( ) is undefined if there have been any calls to
32300 *pthread_cleanup_push*( ) or *pthread_cleanup_pop*( ) made without the matching call since the jump
32301 buffer was filled. The effect of calling *longjmp*( ) or *siglongjmp*( ) from inside a cancelation
32302 cleanup handler is also undefined unless the jump buffer was also filled in the cancelation
32303 cleanup handler.

32304 **RETURN VALUE**

32305 The *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ) functions shall not return a value.

32306 **ERRORS**

32307 No errors are defined.

32308 These functions shall not return an error code of [EINTR].

32309 **EXAMPLES**

32310 The following is an example using thread primitives to implement a cancelable, writers-priority
32311 read-write lock:

```
typedef struct {
    pthread_mutex_t lock;
    pthread_cond_t rcond,
        wcond;
    int lock_count; /* < 0 .. Held by writer. */
                    /* > 0 .. Held by lock_count readers. */
                    /* = 0 .. Held by nobody. */
    int waiting_writers; /* Count of waiting writers. */
} rwlock;

void
waiting_reader_cleanup(void *arg)
{
```
32312
32313
32314
32315
32316
32317
32318
32319
32320
32321
32322
32323

```
32324           rwlock *l;
32325           l = (rwlock *) arg;
32326           pthread_mutex_unlock(&l->lock);
32327       }
32328       void
32329       lock_for_read(rwlock *l)
32330       {
32331           pthread_mutex_lock(&l->lock);
32332           pthread_cleanup_push(waiting_reader_cleanup, l);
32333           while ((l->lock_count < 0) && (l->waiting_writers != 0))
32334               pthread_cond_wait(&l->rcond, &l->lock);
32335           l->lock_count++;
32336          /*
32337           * Note the pthread_cleanup_pop executes
32338           * waiting_reader_cleanup.
32339           */
32340           pthread_cleanup_pop(1);
32341       }
32342       void
32343       release_read_lock(rwlock *l)
32344       {
32345           pthread_mutex_lock(&l->lock);
32346           if (--l->lock_count == 0)
32347               pthread_cond_signal(&l->wcond);
32348           pthread_mutex_unlock(l);
32349       }
32350       void
32351       waiting_writer_cleanup(void *arg)
32352       {
32353           rwlock *l;
32354           l = (rwlock *) arg;
32355           if ((--l->waiting_writers == 0) && (l->lock_count >= 0)) {
32356              /*
32357               * This only happens if we have been canceled.
32358               */
32359               pthread_cond_broadcast(&l->wcond);
32360       }
32361           pthread_mutex_unlock(&l->lock);
32362       }
32363       void
32364       lock_for_write(rwlock *l)
32365       {
32366           pthread_mutex_lock(&l->lock);
32367           l->waiting_writers++;
32368           pthread_cleanup_push(waiting_writer_cleanup, l);
32369           while (l->lock_count != 0)
32370               pthread_cond_wait(&l->wcond, &l->lock);
32371           l->lock_count = -1;
32372          /*
```

```
32373           * Note the pthread_cleanup_pop executes
32374           * waiting_writer_cleanup.
32375           */
32376          pthread_cleanup_pop(1);
32377      }
32378      void
32379      release_write_lock(rwlock *l)
32380      {
32381          pthread_mutex_lock(&l->lock);
32382          l->lock_count = 0;
32383          if (l->waiting_writers == 0)
32384              pthread_cond_broadcast(&l->rcond)
32385          else
32386              pthread_cond_signal(&l->wcond);
32387          pthread_mutex_unlock(&l->lock);
32388      }
32389      /*
32390       * This function is called to initialize the read/write lock.
32391       */
32392      void
32393      initialize_rwlock(rwlock *l)
32394      {
32395          pthread_mutex_init(&l->lock, pthread_mutexattr_default);
32396          pthread_cond_init(&l->wcond, pthread_condattr_default);
32397          pthread_cond_init(&l->rcond, pthread_condattr_default);
32398          l->lock_count = 0;
32399          l->waiting_writers = 0;
32400      }
32401      reader_thread()
32402      {
32403          lock_for_read(&lock);
32404          pthread_cleanup_push(release_read_lock, &lock);
32405         /*
32406          * Thread has read lock.
32407          */
32408          pthread_cleanup_pop(1);
32409      }
32410      writer_thread()
32411      {
32412          lock_for_write(&lock);
32413          pthread_cleanup_push(release_write_lock, &lock);
32414         /*
32415          * Thread has write lock.
32416          */
32417      pthread_cleanup_pop(1);
32418      }
```

APPLICATION USAGE

The two routines that push and pop cancelation cleanup handlers, *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ), can be thought of as left and right parentheses. They always need to be matched.

RATIONALE

The restriction that the two routines that push and pop cancelation cleanup handlers, *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ), have to appear in the same lexical scope allows for efficient macro or compiler implementations and efficient storage management. A sample implementation of these routines as macros might look like this:

```
#define pthread_cleanup_push(rtn,arg) { \
    struct _pthread_handler_rec __cleanup_handler, **__head; \
    __cleanup_handler.rtn = rtn; \
    __cleanup_handler.arg = arg; \
    (void) pthread_getspecific(_pthread_handler_key, &__head); \
    __cleanup_handler.next = *__head; \
    *__head = &__cleanup_handler;

#define pthread_cleanup_pop(ex) \
    *__head = __cleanup_handler.next; \
    if (ex) (*__cleanup_handler.rtn)(__cleanup_handler.arg); \
}
```

A more ambitious implementation of these routines might do even better by allowing the compiler to note that the cancelation cleanup handler is a constant and can be expanded inline.

This volume of IEEE Std 1003.1-200x currently leaves unspecified the effect of calling *longjmp*( ) from a signal handler executing in a POSIX System Interfaces function. If an implementation wants to allow this and give the programmer reasonable behavior, the *longjmp*( ) function has to call all cancelation cleanup handlers that have been pushed but not popped since the time *setjmp*( ) was called.

Consider a multi-threaded function called by a thread that uses signals. If a signal were delivered to a signal handler during the operation of *qsort*( ) and that handler were to call *longjmp*( ) (which, in turn, did *not* call the cancelation cleanup handlers) the helper threads created by the *qsort*( ) function would not be canceled. Instead, they would continue to execute and write into the argument array even though the array might have been popped off of the stack.

Note that the specified cleanup handling mechanism is especially tied to the C language and, while the requirement for a uniform mechanism for expressing cleanup is language-independent, the mechanism used in other languages may be quite different. In addition, this mechanism is really only necessary due to the lack of a real exception mechanism in the C language, which would be the ideal solution.

There is no notion of a cancelation cleanup-safe function. If an application has no cancelation points in its signal handlers, blocks any signal whose handler may have cancelation points while calling async-unsafe functions, or disables cancelation while calling async-unsafe functions, all functions may be safely called from cancelation cleanup routines.

FUTURE DIRECTIONS

None.

32463 **SEE ALSO**

32464 *pthread_cancel*( ), *pthread_setcancelstate*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
32465 **<pthread.h>**

32466 **CHANGE HISTORY**

32467 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32468 **Issue 6**

32469 The *pthread_cleanup_pop*( ) and *pthread_cleanup_push*( ) functions are marked as part of the
32470 Threads option.

32471 The APPLICATION USAGE section is added.

32472 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

pthread_cond_broadcast, pthread_cond_signal — broadcast or signal a condition

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

DESCRIPTION

These functions shall unblock threads blocked on a condition variable.

The *pthread_cond_broadcast*() function shall unblock all threads currently blocked on the specified condition variable *cond*.

The *pthread_cond_signal*() function shall unblock at least one of the threads that are blocked on the specified condition variable *cond* (if any threads are blocked on *cond*).

If more than one thread is blocked on a condition variable, the scheduling policy shall determine the order in which threads are unblocked. When each thread unblocked as a result of a *pthread_cond_broadcast*() or *pthread_cond_signal*() returns from its call to *pthread_cond_wait*() or *pthread_cond_timedwait*(), the thread shall own the mutex with which it called *pthread_cond_wait*() or *pthread_cond_timedwait*(). The thread(s) that are unblocked shall contend for the mutex according to the scheduling policy (if applicable), and as if each had called *pthread_mutex_lock*().

The *pthread_cond_broadcast*() or *pthread_cond_signal*() functions may be called by a thread whether or not it currently owns the mutex that threads calling *pthread_cond_wait*() or *pthread_cond_timedwait*() have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex shall be locked by the thread calling *pthread_cond_broadcast*() or *pthread_cond_signal*().

The *pthread_cond_broadcast*() and *pthread_cond_signal*() functions shall have no effect if there are no threads currently blocked on *cond*.

RETURN VALUE

If successful, the *pthread_cond_broadcast*() and *pthread_cond_signal*() functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_cond_broadcast*() and *pthread_cond_signal*() function may fail if:

[EINVAL] The value *cond* does not refer to an initialized condition variable.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_cond_broadcast*() function is used whenever the shared-variable state has been changed in a way that more than one thread can proceed with its task. Consider a single producer/multiple consumer problem, where the producer can insert multiple items on a list that is accessed one item at a time by the consumers. By calling the *pthread_cond_broadcast*() function, the producer would notify all consumers that might be waiting, and thereby the application would receive more throughput on a multi-processor. In addition, *pthread_cond_broadcast*() makes it easier to implement a read-write lock. The *pthread_cond_broadcast*() function is needed in order to wake up all waiting readers when a

writer releases its lock. Finally, the two-phase commit algorithm can use this broadcast function to notify all clients of an impending transaction commit.

It is not safe to use the *pthread_cond_signal*() function in a signal handler that is invoked asynchronously. Even if it were safe, there would still be a race between the test of the Boolean *pthread_cond_wait*() that could not be efficiently eliminated.

Mutexes and condition variables are thus not suitable for releasing a waiting thread by signaling from code running in a signal handler.

RATIONALE

Multiple Awakenings by Condition Signal

On a multi-processor, it may be impossible for an implementation of *pthread_cond_signal*() to avoid the unblocking of more than one thread blocked on a condition variable. For example, consider the following partial implementation of *pthread_cond_wait*() and *pthread_cond_signal*(), executed by two threads in the order given. One thread is trying to wait on the condition variable, another is concurrently executing *pthread_cond_signal*(), while a third thread is already waiting.

```
pthread_cond_wait(mutex, cond):
    value = cond->value; /* 1 */
    pthread_mutex_unlock(mutex); /* 2 */
    pthread_mutex_lock(cond->mutex); /* 10 */
    if (value == cond->value) { /* 11 */
        me->next_cond = cond->waiter;
        cond->waiter = me;
        pthread_mutex_unlock(cond->mutex);
        unable_to_run(me);
    } else
        pthread_mutex_unlock(cond->mutex); /* 12 */
    pthread_mutex_lock(mutex); /* 13 */

pthread_cond_signal(cond):
    pthread_mutex_lock(cond->mutex); /* 3 */
    cond->value++; /* 4 */
    if (cond->waiter) { /* 5 */
        sleeper = cond->waiter; /* 6 */
        cond->waiter = sleeper->next_cond; /* 7 */
        able_to_run(sleeper); /* 8 */
    }
    pthread_mutex_unlock(cond->mutex); /* 9 */
```

The effect is that more than one thread can return from its call to *pthread_cond_wait*() or *pthread_cond_timedwait*() as a result of one call to *pthread_cond_signal*(). This effect is called ‘‘spurious wakeup’’. Note that the situation is self-correcting in that the number of threads that are so awakened is finite; for example, the next thread to call *pthread_cond_wait*() after the sequence of events above blocks.

While this problem could be resolved, the loss of efficiency for a fringe condition that occurs only rarely is unacceptable, especially given that one has to check the predicate associated with a condition variable anyway. Correcting this problem would unnecessarily reduce the degree of concurrency in this basic building block for all higher-level synchronization operations.

An added benefit of allowing spurious wakeups is that applications are forced to code a predicate-testing-loop around the condition wait. This also makes the application tolerate

32565 superfluous condition broadcasts or signals on the same condition variable that may be coded in
32566 some other part of the application. The resulting applications are thus more robust. Therefore,
32567 IEEE Std 1003.1-200x explicitly documents that spurious wakeups may occur.

32568 **FUTURE DIRECTIONS**

32569 None.

32570 **SEE ALSO**

32571 *pthread_cond_destroy*(), *pthread_cond_timedwait*(), *pthread_cond_wait*(), the Base Definitions
32572 volume of IEEE Std 1003.1-200x, **<pthread.h>**

32573 **CHANGE HISTORY**

32574 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32575 **Issue 6**

32576 The *pthread_cond_broadcast*() and *pthread_cond_signal*() functions are marked as part of the
32577 Threads option.

32578 The APPLICATION USAGE section is added.

NAME

pthread_cond_destroy, pthread_cond_init — destroy and initialize condition variables

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
    const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

DESCRIPTION

The *pthread_cond_destroy*( ) function shall destroy the given condition variable specified by *cond*; the object becomes, in effect, uninitialized. An implementation may cause *pthread_cond_destroy*( ) to set the object referenced by *cond* to an invalid value. A destroyed condition variable object can be reinitialized using *pthread_cond_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

The *pthread_cond_init*( ) function shall initialize the condition variable referenced by *cond* with attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes shall be used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable shall become initialized.

Only *cond* itself may be used for performing synchronization. The result of referring to copies of *cond* in calls to *pthread_cond_wait*( ), *pthread_cond_timedwait*( ), *pthread_cond_signal*( ), *pthread_cond_broadcast*( ), and *pthread_cond_destroy*( ) is undefined.

Attempting to initialize an already initialized condition variable results in undefined behavior.

In cases where default condition variable attributes are appropriate, the macro PTHREAD_COND_INITIALIZER can be used to initialize condition variables that are statically allocated. The effect shall be equivalent to dynamic initialization by a call to *pthread_cond_init*( ) with parameter *attr* specified as NULL, except that no error checks are performed.

RETURN VALUE

If successful, the *pthread_cond_destroy*( ) and *pthread_cond_init*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

The [EBUSY] and [EINVAL] error checks, if implemented, shall act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the condition variable specified by *cond*.

ERRORS

The *pthread_cond_destroy*( ) function may fail if:

[EBUSY] The implementation has detected an attempt to destroy the object referenced by *cond* while it is referenced (for example, while being used in a *pthread_cond_wait*( ) or *pthread_cond_timedwait*( )) by another thread.

[EINVAL] The value specified by *cond* is invalid.

The *pthread_cond_init*( ) function shall fail if:

[EAGAIN] The system lacked the necessary resources (other than memory) to initialize another condition variable.

[ENOMEM] Insufficient memory exists to initialize the condition variable.

The *pthread_cond_init*( ) function may fail if:

[EBUSY] The implementation has detected an attempt to reinitialize the object referenced by *cond*, a previously initialized, but not yet destroyed, condition variable. |

[EINVAL] The value specified by *attr* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

A condition variable can be destroyed immediately after all the threads that are blocked on it are awakened. For example, consider the following code:

```
struct list {
    pthread_mutex_t lm;
    ...
}

struct elt {
    key k;
    int busy;
    pthread_cond_t notbusy;
    ...
}

/* Find a list element and reserve it. */
struct elt *
list_find(struct list *lp, key k)
{
    struct elt *ep;

    pthread_mutex_lock(&lp->lm);
    while ((ep = find_elt(l, k) != NULL) && ep->busy)
        pthread_cond_wait(&ep->notbusy, &lp->lm);
    if (ep != NULL)
        ep->busy = 1;
    pthread_mutex_unlock(&lp->lm);
    return(ep);
}

delete_elt(struct list *lp, struct elt *ep)
{
    pthread_mutex_lock(&lp->lm);
    assert(ep->busy);
    ... remove ep from list ...
    ep->busy = 0;  /* Paranoid. */
(A) pthread_cond_broadcast(&ep->notbusy);
    pthread_mutex_unlock(&lp->lm);
(B) pthread_cond_destroy(&rp->notbusy);
    free(ep);
}
```

In this example, the condition variable and its list element may be freed (line B) immediately after all threads waiting for it are awakened (line A), since the mutex and the code ensure that no other thread can touch the element to be deleted.

32671 **APPLICATION USAGE**

32672 None.

32673 **RATIONALE**

32674 See *pthread_mutex_init*( ); a similar rationale applies to condition variables.

32675 **FUTURE DIRECTIONS**

32676 None.

32677 **SEE ALSO**

32678 *pthread_cond_broadcast*( ), *pthread_cond_signal*( ), *pthread_cond_timedwait*( ), *pthread_cond_wait*( ),
32679 the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

32680 **CHANGE HISTORY**

32681 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32682 **Issue 6**

32683 The *pthread_cond_destroy*( ) and *pthread_cond_init*( ) functions are marked as part of the Threads
32684 option.

32685 IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

32686 The **restrict** keyword is added to the *pthread_cond_init*( ) prototype for alignment with the
32687 ISO / IEC 9899: 1999 standard.

32688 **NAME**

32689 pthread_cond_init — initialize condition variables

32690 **SYNOPSIS**

32691 THR
```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *restrict cond,
    const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```
32695

32696 **DESCRIPTION**

32697 Refer to *pthread_cond_destroy*( ).

32698 **NAME**

32699 pthread_cond_signal — signal a condition

32700 **SYNOPSIS**

32701 THR
```
#include <pthread.h>
```

32702
```
int pthread_cond_signal(pthread_cond_t *cond);
```

32703

32704 **DESCRIPTION**

32705 Refer to *pthread_cond_broadcast*( ).

NAME

pthread_cond_timedwait, pthread_cond_wait — wait on a condition

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
```

DESCRIPTION

The *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) functions shall block on a condition variable. They shall be called with *mutex* locked by the calling thread or undefined behavior results.

These functions atomically release *mutex* and cause the calling thread to block on the condition variable *cond*; atomically here means ''atomically with respect to access by another thread to the mutex and then the condition variable''. That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ) in that thread shall behave as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.

When using condition variables there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) functions may occur. Since the return from *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return.

The effect of using more than one mutex for concurrent *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) operations on the same condition variable is undefined; that is, a condition variable becomes bound to a unique mutex when a thread waits on the condition variable, and this (dynamic) binding shall end when the wait returns.

A condition wait (whether timed or not) is a cancelation point. When the cancelability enable state of a thread is set to PTHREAD_CANCEL_DEFERRED, a side effect of acting upon a cancelation request while in a condition wait is that the mutex is (in effect) re-acquired before calling the first cancelation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ), but at that point notices the cancelation request and instead of returning to the caller of *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ), starts the thread cancelation activities, which includes calling cancelation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) shall not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The *pthread_cond_timedwait*( ) function shall be equivalent to *pthread_cond_wait*( ), except that an error is returned if the absolute time specified by *abstime* passes (that is, system time equals or exceeds *abstime*) before the condition *cond* is signaled or broadcasted, or if the absolute time

CS

specified by *abstime* has already been passed at the time of the call. If the Clock Selection option is supported, the condition variable shall have a clock attribute which specifies the clock that shall be used to measure the time specified by the *abstime* argument. When such timeouts occur, *pthread_cond_timedwait*( ) shall nonetheless release and re-acquire the mutex referenced by *mutex*. The *pthread_cond_timedwait*( ) function is also a cancelation point.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it shall return zero due to spurious wakeup.

RETURN VALUE

Except in the case of [ETIMEDOUT], all these error checks shall act as if they were performed immediately at the beginning of processing for the function and shall cause an error return, in effect, prior to modifying the state of the mutex specified by *mutex* or the condition variable specified by *cond*.

Upon successful completion, a value of zero shall be returned; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_cond_timedwait*( ) function shall fail if:

[ETIMEDOUT] The time specified by *abstime* to *pthread_cond_timedwait*( ) has passed.

The *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) functions may fail if:

[EINVAL] The value specified by *cond*, *mutex*, or *abstime* is invalid.

[EINVAL] Different mutexes were supplied for concurrent *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) operations on the same condition variable.

[EPERM] The mutex was not owned by the current thread at the time of the call.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Condition Wait Semantics

It is important to note that when *pthread_cond_wait*( ) and *pthread_cond_timedwait*( ) return without error, the associated predicate may still be false. Similarly, when *pthread_cond_timedwait*( ) returns with the timeout error, the associated predicate may be true due to an unavoidable race between the expiration of the timeout and the predicate state change.

Some implementations, particularly on a multi-processor, may sometimes cause multiple threads to wake up when the condition variable is signaled simultaneously on different processors.

In general, whenever a condition wait returns, the thread has to re-evaluate the predicate associated with the condition wait to determine whether it can safely proceed, should wait again, or should declare a timeout. A return from the wait does not imply that the associated predicate is either true or false.

It is thus recommended that a condition wait be enclosed in the equivalent of a ‘‘while loop’’ that checks the predicate.

Timed Wait Semantics

An absolute time measure was chosen for specifying the timeout parameter for two reasons. First, a relative time measure can be easily implemented on top of a function that specifies absolute time, but there is a race condition associated with specifying an absolute timeout on top of a function that specifies relative timeouts. For example, assume that *clock_gettime*( ) returns the current time and *cond_relative_timed_wait*( ) uses relative timeouts:

```
clock_gettime(CLOCK_REALTIME, &now)
reltime = sleep_til_this_absolute_time -now;
cond_relative_timed_wait(c, m, &reltime);
```

If the thread is preempted between the first statement and the last statement, the thread blocks for too long. Blocking, however, is irrelevant if an absolute timeout is used. An absolute timeout also need not be recomputed if it is used multiple times in a loop, such as that enclosing a condition wait.

For cases when the system clock is advanced discontinuously by an operator, it is expected that implementations process any timed wait expiring at an intervening time as if that time had actually occurred.

Cancelation and Condition Wait

A condition wait, whether timed or not, is a cancelation point. That is, the functions *pthread_cond_wait*( ) or *pthread_cond_timedwait*( ) are points where a pending (or concurrent) cancelation request is noticed. The reason for this is that an indefinite wait is possible at these points—whatever event is being waited for, even if the program is totally correct, might never occur; for example, some input data being awaited might never be sent. By making condition wait a cancelation point, the thread can be canceled and perform its cancelation cleanup handler even though it may be stuck in some indefinite wait.

A side effect of acting on a cancelation request while a thread is blocked on a condition variable is to re-acquire the mutex before calling any of the cancelation cleanup handlers. This is done in order to ensure that the cancelation cleanup handler is executed in the same state as the critical code that lies both before and after the call to the condition wait function. This rule is also required when interfacing to POSIX threads from languages, such as Ada or C++, which may choose to map cancelation onto a language exception; this rule ensures that each exception handler guarding a critical section can always safely depend upon the fact that the associated mutex has already been locked regardless of exactly where within the critical section the exception was raised. Without this rule, there would not be a uniform rule that exception handlers could follow regarding the lock, and so coding would become very cumbersome.

Therefore, since *some* statement has to be made regarding the state of the lock when a cancelation is delivered during a wait, a definition has been chosen that makes application coding most convenient and error free.

When acting on a cancelation request while a thread is blocked on a condition variable, the implementation is required to ensure that the thread does not consume any condition signals directed at that condition variable if there are any other threads waiting on that condition variable. This rule is specified in order to avoid deadlock conditions that could occur if these two independent requests (one acting on a thread and the other acting on the condition variable) were not processed independently.

32839 **Performance of Mutexes and Condition Variables**

32840 Mutexes are expected to be locked only for a few instructions. This practice is almost
32841 automatically enforced by the desire of programmers to avoid long serial regions of execution
32842 (which would reduce total effective parallelism).

32843 When using mutexes and condition variables, one tries to ensure that the usual case is to lock the
32844 mutex, access shared data, and unlock the mutex. Waiting on a condition variable should be a
32845 relatively rare situation. For example, when implementing a read-write lock, code that acquires a
32846 read-lock typically needs only to increment the count of readers (under mutual-exclusion) and
32847 return. The calling thread would actually wait on the condition variable only when there is
32848 already an active writer. So the efficiency of a synchronization operation is bounded by the cost
32849 of mutex lock/unlock and not by condition wait. Note that in the usual case there is no context
32850 switch.

32851 This is not to say that the efficiency of condition waiting is unimportant. Since there needs to be
32852 at least one context switch per Ada rendezvous, the efficiency of waiting on a condition variable
32853 is important. The cost of waiting on a condition variable should be little more than the minimal
32854 cost for a context switch plus the time to unlock and lock the mutex.

32855 **Features of Mutexes and Condition Variables**

32856 It had been suggested that the mutex acquisition and release be decoupled from condition wait.
32857 This was rejected because it is the combined nature of the operation that, in fact, facilitates
32858 realtime implementations. Those implementations can atomically move a high-priority thread
32859 between the condition variable and the mutex in a manner that is transparent to the caller. This
32860 can prevent extra context switches and provide more deterministic acquisition of a mutex when
32861 the waiting thread is signaled. Thus, fairness and priority issues can be dealt with directly by the
32862 scheduling discipline. Furthermore, the current condition wait operation matches existing
32863 practice.

32864 **Scheduling Behavior of Mutexes and Condition Variables**

32865 Synchronization primitives that attempt to interfere with scheduling policy by specifying an
32866 ordering rule are considered undesirable. Threads waiting on mutexes and condition variables
32867 are selected to proceed in an order dependent upon the scheduling policy rather than in some
32868 fixed order (for example, FIFO or priority). Thus, the scheduling policy determines which
32869 thread(s) are awakened and allowed to proceed.

32870 **Timed Condition Wait**

32871 The *pthread_cond_timedwait*( ) function allows an application to give up waiting for a particular
32872 condition after a given amount of time. An example of its use follows:

```
32873 (void) pthread_mutex_lock(&t.mn);
32874         t.waiters++;
32875     clock_gettime(CLOCK_REALTIME, &ts);
32876     ts.tv_sec += 5;
32877     rc = 0;
32878     while (! mypredicate(&t) && rc == 0)
32879         rc = pthread_cond_timedwait(&t.cond, &t.mn, &ts);
32880     t.waiters--;
32881     if (rc == 0) setmystate(&t);
32882 (void) pthread_mutex_unlock(&t.mn);
```

By making the timeout parameter absolute, it does not need to be recomputed each time the program checks its blocking predicate. If the timeout was relative, it would have to be recomputed before each call. This would be especially difficult since such code would need to take into account the possibility of extra wakeups that result from extra broadcasts or signals on the condition variable that occur before either the predicate is true or the timeout is due.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_signal( ), *pthread_cond_broadcast*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) functions are marked as part of the Threads option.

The Open Group Corrigendum U021/9 is applied, correcting the prototype for the *pthread_cond_wait*( ) function.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for the Clock Selection option.

The ERRORS section has an additional case for [EPERM] in response to IEEE PASC Interpretation 1003.1c #28.

The **restrict** keyword is added to the *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) prototypes for alignment with the ISO/IEC 9899: 1999 standard.

32906 **NAME**

32907 pthread_cond_wait — wait on a condition

32908 **SYNOPSIS**

32909 THR
```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
```
32912

32913 **DESCRIPTION**

32914 Refer to *pthread_cond_timedwait*( ).

32915 **NAME**

32916 pthread_condattr_destroy, pthread_condattr_init — destroy and initialize condition variable
32917 attributes object

32918 **SYNOPSIS**

32919 THR
```
#include <pthread.h>

int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_condattr_init(pthread_condattr_t *attr);
```
32922

32923 **DESCRIPTION**

32924 The *pthread_condattr_destroy*( ) function shall destroy a condition variable attributes object; the
32925 object becomes, in effect, uninitialized. An implementation may cause *pthread_condattr_destroy*( )
32926 to set the object referenced by *attr* to an invalid value. A destroyed *attr* attributes object can be |
32927 reinitialized using *pthread_condattr_init*( ); the results of otherwise referencing the object after it |
32928 has been destroyed are undefined. |

32929 The *pthread_condattr_init*( ) function shall initialize a condition variable attributes object *attr* with
32930 the default value for all of the attributes defined by the implementation.

32931 Results are undefined if *pthread_condattr_init*( ) is called specifying an already initialized *attr* |
32932 attributes object. |

32933 After a condition variable attributes object has been used to initialize one or more condition
32934 variables, any function affecting the attributes object (including destruction) shall not affect any |
32935 previously initialized condition variables.

32936 This volume of IEEE Std 1003.1-200x requires two attributes, the *clock* attribute and the *process-*
32937 *shared* attribute.

32938 Additional attributes, their default values, and the names of the associated functions to get and
32939 set those attribute values are implementation-defined.

32940 **RETURN VALUE**

32941 If successful, the *pthread_condattr_destroy*( ) and *pthread_condattr_init*( ) functions shall return
32942 zero; otherwise, an error number shall be returned to indicate the error.

32943 **ERRORS**

32944 The *pthread_condattr_destroy*( ) function may fail if:

32945 [EINVAL] The value specified by *attr* is invalid.

32946 The *pthread_condattr_init*( ) function shall fail if:

32947 [ENOMEM] Insufficient memory exists to initialize the condition variable attributes object.

32948 These functions shall not return an error code of [EINTR].

32949 **EXAMPLES**

32950 None.

32951 **APPLICATION USAGE**

32952 None.

32953 **RATIONALE**

32954 See *pthread_attr_init*( ) and *pthread_mutex_init*( ).

32955 A *process-shared* attribute has been defined for condition variables for the same reason it has been
32956 defined for mutexes.

32957 **FUTURE DIRECTIONS**

32958 None.

32959 **SEE ALSO**

32960 *pthread_cond_destroy*( ), *pthread_condattr_getpshared*( ), *pthread_create*( ), *pthread_mutex_destroy*( ),
32961 the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

32962 **CHANGE HISTORY**

32963 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32964 **Issue 6**

32965 The *pthread_condattr_destroy*( ) and *pthread_condattr_init*( ) functions are marked as part of the
32966 Threads option.

NAME

pthread_condattr_getclock, pthread_condattr_setclock — get and set the clock selection condition variable attribute (**ADVANCED REALTIME**)

SYNOPSIS

THR CS
```
#include <pthread.h>

int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
    clockid_t *restrict clock_id);
int pthread_condattr_setclock(pthread_condattr_t *attr,
    clockid_t clock_id);
```

DESCRIPTION

The *pthread_condattr_getclock*( ) function shall obtain the value of the *clock* attribute from the attributes object referenced by *attr*. The *pthread_condattr_setclock*( ) function shall set the *clock* attribute in an initialized attributes object referenced by *attr*. If *pthread_condattr_setclock*( ) is called with a *clock_id* argument that refers to a CPU-time clock, the call shall fail.

The *clock* attribute is the clock ID of the clock that shall be used to measure the timeout service of *pthread_cond_timedwait*( ). The default value of the *clock* attribute shall refer to the system clock.

RETURN VALUE

If successful, the *pthread_condattr_getclock*( ) function shall return zero and store the value of the clock attribute of *attr* into the object referenced by the *clock_id* argument. Otherwise, an error number shall be returned to indicate the error.

If successful, the *pthread_condattr_setclock*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL] The value specified by *attr* is invalid.

The *pthread_condattr_setclock*( ) function may fail if:

[EINVAL] The value specified by *clock_id* does not refer to a known clock, or is a CPU-time clock.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_init( ), *pthread_cond_timedwait*( ), *pthread_condattr_destroy*( ), *pthread_condattr_getpshared*( ) (on page 1536),1 *pthread_condattr_init*( ), *pthread_condattr_setpshared*( ) (on page 1540),1 *pthread_create*( ), *pthread_mutex_init*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

33010 **CHANGE HISTORY**

33011 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

33012 **NAME**

33013 pthread_condattr_getpshared, pthread_condattr_setpshared — get and set the process-shared
33014 condition variable attributes

33015 **SYNOPSIS**

33016 THR TSH
```
#include <pthread.h>

int pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,
    int *restrict pshared);
int pthread_condattr_setpshared(pthread_condattr_t *attr,
    int pshared);
```
33021

33022 **DESCRIPTION**

33023 The *pthread_condattr_getpshared*( ) function shall obtain the value of the *process-shared* attribute
33024 from the attributes object referenced by *attr*. The *pthread_condattr_setpshared*( ) function shall set |
33025 the *process-shared* attribute in an initialized attributes object referenced by *attr*. |

33026 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a condition
33027 variable to be operated upon by any thread that has access to the memory where the condition
33028 variable is allocated, even if the condition variable is allocated in memory that is shared by
33029 multiple processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the
33030 condition variable shall only be operated upon by threads created within the same process as the |
33031 thread that initialized the condition variable; if threads of differing processes attempt to operate |
33032 on such a condition variable, the behavior is undefined. The default value of the attribute is |
33033 PTHREAD_PROCESS_PRIVATE.

33034 **RETURN VALUE**

33035 If successful, the *pthread_condattr_setpshared*( ) function shall return zero; otherwise, an error
33036 number shall be returned to indicate the error.

33037 If successful, the *pthread_condattr_getpshared*( ) function shall return zero and store the value of
33038 the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise,
33039 an error number shall be returned to indicate the error.

33040 **ERRORS**

33041 The *pthread_condattr_getpshared*( ) and *pthread_condattr_setpshared*( ) functions may fail if:

33042 [EINVAL] The value specified by *attr* is invalid.

33043 The *pthread_condattr_setpshared*( ) function may fail if:

33044 [EINVAL] The new value specified for the attribute is outside the range of legal values
33045 for that attribute.

33046 These functions shall not return an error code of [EINTR].

33047 **EXAMPLES**

33048 None.

33049 **APPLICATION USAGE**

33050 None.

33051 **RATIONALE**

33052 None.

33053 **FUTURE DIRECTIONS**

33054 None.

33055 **SEE ALSO**

33056 *pthread_create*( ), *pthread_cond_destroy*( ), *pthread_condattr_destroy*( ), *pthread_mutex_destroy*( ), the
33057 Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

33058 **CHANGE HISTORY**

33059 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33060 **Issue 6**

33061 The *pthread_condattr_getpshared*( ) and *pthread_condattr_setpshared*( ) functions are marked as part
33062 of the Threads and Thread Process-Shared Synchronization options.

33063 The **restrict** keyword is added to the *pthread_condattr_getpshared*( ) prototype for alignment with
33064 the ISO / IEC 9899: 1999 standard.

33065 **NAME**

33066 pthread_condattr_init — initialize condition variable attributes object

33067 **SYNOPSIS**

33068 THR
```
#include <pthread.h>
```

33069
```
int pthread_condattr_init(pthread_condattr_t *attr);
```

33070

33071 **DESCRIPTION**

33072 Refer to *pthread_condattr_destroy*( ).

33073 **NAME**

33074 pthread_condattr_setclock — set the clock selection condition variable attribute

33075 **SYNOPSIS**

33076 THR CS
```
#include <pthread.h>

int pthread_condattr_setclock(pthread_condattr_t *attr,
    clockid_t clock_id);
```
33079

33080 **DESCRIPTION**

33081 Refer to *pthread_condattr_getclock*( ).

33082 **NAME**

33083 pthread_condattr_setpshared — set the process-shared condition variable attributes

33084 **SYNOPSIS**

33085 THR TSH
```
#include <pthread.h>

int pthread_condattr_setpshared(pthread_condattr_t *attr,
    int pshared);
```
33086
33087
33088

33089 **DESCRIPTION**

33090 Refer to *pthread_condattr_getpshared*( ).

33091 **NAME**

33092 pthread_create — thread creation

33093 **SYNOPSIS**

33094 THR
```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
    const pthread_attr_t *restrict attr,
    void *(*start_routine)(void*), void *restrict arg);
```
33098

33099 **DESCRIPTION**

33100 The *pthread_create*( ) function shall create a new thread, with attributes specified by *attr*, within a
33101 process. If *attr* is NULL, the default attributes shall be used. If the attributes specified by *attr* are
33102 modified later, the thread's attributes shall not be affected. Upon successful completion,
33103 *pthread_create*( ) shall store the ID of the created thread in the location referenced by *thread*.

33104 The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine*
33105 returns, the effect shall be as if there was an implicit call to *pthread_exit*( ) using the return value
33106 of *start_routine* as the exit status. Note that the thread in which *main*( ) was originally invoked
33107 differs from this. When it returns from *main*( ), the effect shall be as if there was an implicit call
33108 to *exit*( ) using the return value of *main*( ) as the exit status.

33109 The signal state of the new thread shall be initialized as follows:

33110 - The signal mask shall be inherited from the creating thread.

33111 - The set of signals pending for the new thread shall be empty.

33112 The floating-point environment shall be inherited from the creating thread.

33113 If *pthread_create*( ) fails, no new thread is created and the contents of the location referenced by
33114 *thread* are undefined.

33115 TCT If _POSIX_THREAD_CPUTIME is defined, the new thread shall have a CPU-time clock
33116 accessible, and the initial value of this clock shall be set to zero.

33117 **RETURN VALUE**

33118 If successful, the *pthread_create*( ) function shall return zero; otherwise, an error number shall be
33119 returned to indicate the error.

33120 **ERRORS**

33121 The *pthread_create*( ) function shall fail if:

33122 [EAGAIN] The system lacked the necessary resources to create another thread, or the
33123 system-imposed limit on the total number of threads in a process
33124 PTHREAD_THREADS_MAX would be exceeded.

33125 [EINVAL] The value specified by *attr* is invalid.

33126 [EPERM] The caller does not have appropriate permission to set the required
33127 scheduling parameters or scheduling policy.

33128 The *pthread_create*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

A suggested alternative to *pthread_create*( ) would be to define two separate operations: create and start. Some applications would find such behavior more natural. Ada, in particular, separates the ‘‘creation’’ of a task from its ‘‘activation’’.

Splitting the operation was rejected by the standard developers for many reasons:

- The number of calls required to start a thread would increase from one to two and thus place an additional burden on applications that do not require the additional synchronization. The second call, however, could be avoided by the additional complication of a start-up state attribute.
- An extra state would be introduced: ‘‘created but not started’’. This would require the standard to specify the behavior of the thread operations when the target has not yet started executing.
- For those applications that require such behavior, it is possible to simulate the two separate steps with the facilities that are currently provided. The *start_routine*( ) can synchronize by waiting on a condition variable that is signaled by the start operation.

An Ada implementor can choose to create the thread at either of two points in the Ada program: when the task object is created, or when the task is activated (generally at a ‘‘begin’’). If the first approach is adopted, the *start_routine*( ) needs to wait on a condition variable to receive the order to begin ‘‘activation’’. The second approach requires no such condition variable or extra synchronization. In either approach, a separate Ada task control block would need to be created when the task object is created to hold rendezvous queues, and so on.

An extension of the preceding model would be to allow the state of the thread to be modified between the create and start. This would allow the thread attributes object to be eliminated. This has been rejected because:

- All state in the thread attributes object has to be able to be set for the thread. This would require the definition of functions to modify thread attributes. There would be no reduction in the number of function calls required to set up the thread. In fact, for an application that creates all threads using identical attributes, the number of function calls required to set up the threads would be dramatically increased. Use of a thread attributes object permits the application to make one set of attribute setting function calls. Otherwise, the set of attribute setting function calls needs to be made for each thread creation.
- Depending on the implementation architecture, functions to set thread state would require kernel calls, or for other implementation reasons would not be able to be implemented as macros, thereby increasing the cost of thread creation.
- The ability for applications to segregate threads by class would be lost.

Another suggested alternative uses a model similar to that for process creation, such as ‘‘thread fork’’. The fork semantics would provide more flexibility and the ‘‘create’’ function can be implemented simply by doing a thread fork followed immediately by a call to the desired ‘‘start routine’’ for the thread. This alternative has these problems:

- For many implementations, the entire stack of the calling thread would need to be duplicated, since in many architectures there is no way to determine the size of the calling frame.

- Efficiency is reduced since at least some part of the stack has to be copied, even though in most cases the thread never needs the copied context, since it merely calls the desired start routine.

FUTURE DIRECTIONS

None.

SEE ALSO

fork( ), *pthread_exit*( ), *pthread_join*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_create*( ) function is marked as part of the Threads option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EPERM] mandatory error condition is added.

The thread CPU-time clock semantics are added for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *pthread_create*( ) prototype for alignment with the ISO/IEC 9899:1999 standard. |

The DESCRIPTION is updated to make it explicit that the floating-point environment is inherited from the creating thread. | |

NAME

pthread_detach — detach a thread

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

DESCRIPTION

The *pthread_detach*( ) function shall indicate to the implementation that storage for the thread *thread* can be reclaimed when that thread terminates. If *thread* has not terminated, *pthread_detach*( ) shall not cause it to terminate. The effect of multiple *pthread_detach*( ) calls on the same target thread is unspecified.

RETURN VALUE

If the call succeeds, *pthread_detach*( ) shall return 0; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_detach*( ) function shall fail if:

[EINVAL] The implementation has detected that the value specified by *thread* does not refer to a joinable thread.

[ESRCH] No thread could be found corresponding to that specified by the given thread ID.

The *pthread_detach*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *pthread_join*( ) or *pthread_detach*( ) functions should eventually be called for every thread that is created so that storage associated with the thread may be reclaimed.

It has been suggested that a ''detach'' function is not necessary; the *detachstate* thread creation attribute is sufficient, since a thread need never be dynamically detached. However, need arises in at least two cases:

1. In a cancelation handler for a *pthread_join*( ) it is nearly essential to have a *pthread_detach*( ) function in order to detach the thread on which *pthread_join*( ) was waiting. Without it, it would be necessary to have the handler do another *pthread_join*( ) to attempt to detach the thread, which would both delay the cancelation processing for an unbounded period and introduce a new call to *pthread_join*( ), which might itself need a cancelation handler. A dynamic detach is nearly essential in this case.
2. In order to detach the ''initial thread'' (as may be desirable in processes that set up server threads).

FUTURE DIRECTIONS

None.

33236 **SEE ALSO**

33237 *pthread_join*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

33238 **CHANGE HISTORY**

33239 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33240 **Issue 6**

33241 The *pthread_detach*( ) function is marked as part of the Threads option.

33242 **NAME**

33243 pthread_equal — compare thread IDs

33244 **SYNOPSIS**

33245 THR
```
#include <pthread.h>
```
33246
```
int pthread_equal(pthread_t t1, pthread_t t2);
```
33247

33248 **DESCRIPTION**

33249 This function shall compare the thread IDs *t1* and *t2*.

33250 **RETURN VALUE**

33251 The *pthread_equal*( ) function shall return a non-zero value if *t1* and *t2* are equal; otherwise, zero
33252 shall be returned.

33253 If either *t1* or *t2* are not valid thread IDs, the behavior is undefined.

33254 **ERRORS**

33255 No errors are defined.

33256 The *pthread_equal*( ) function shall not return an error code of [EINTR].

33257 **EXAMPLES**

33258 None.

33259 **APPLICATION USAGE**

33260 None.

33261 **RATIONALE**

33262 Implementations may choose to define a thread ID as a structure. This allows additional
33263 flexibility and robustness over using an **int**. For example, a thread ID could include a sequence
33264 number that allows detection of ‘‘dangling IDs’’ (copies of a thread ID that has been detached).
33265 Since the C language does not support comparison on structure types, the *pthread_equal*( )
33266 function is provided to compare thread IDs.

33267 **FUTURE DIRECTIONS**

33268 None.

33269 **SEE ALSO**

33270 *pthread_create*( ), *pthread_self*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

33271 **CHANGE HISTORY**

33272 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33273 **Issue 6**

33274 The *pthread_equal*( ) function is marked as part of the Threads option.

33275 **NAME**

33276 pthread_exit — thread termination

33277 **SYNOPSIS**

33278 THR
```
#include <pthread.h>
```

33279
```
void pthread_exit(void *value_ptr);
```

33280

33281 **DESCRIPTION**

33282 The *pthread_exit*( ) function shall terminate the calling thread and make the value *value_ptr*
33283 available to any successful join with the terminating thread. Any cancelation cleanup handlers
33284 that have been pushed and not yet popped shall be popped in the reverse order that they were
33285 pushed and then executed. After all cancelation cleanup handlers have been executed, if the
33286 thread has any thread-specific data, appropriate destructor functions shall be called in an
33287 unspecified order. Thread termination does not release any application visible process resources,
33288 including, but not limited to, mutexes and file descriptors, nor does it perform any process-level
33289 cleanup actions, including, but not limited to, calling any *atexit*( ) routines that may exist.

33290 An implicit call to *pthread_exit*( ) is made when a thread other than the thread in which *main*( )
33291 was first invoked returns from the start routine that was used to create it. The function's return
33292 value shall serve as the thread's exit status.

33293 The behavior of *pthread_exit*( ) is undefined if called from a cancelation cleanup handler or
33294 destructor function that was invoked as a result of either an implicit or explicit call to
33295 *pthread_exit*( ).

33296 After a thread has terminated, the result of access to local (auto) variables of the thread is
33297 undefined. Thus, references to local variables of the exiting thread should not be used for the
33298 *pthread_exit*( ) *value_ptr* parameter value.

33299 The process shall exit with an exit status of 0 after the last thread has been terminated. The
33300 behavior shall be as if the implementation called *exit*( ) with a zero argument at thread
33301 termination time.

33302 **RETURN VALUE**

33303 The *pthread_exit*( ) function cannot return to its caller.

33304 **ERRORS**

33305 No errors are defined.

33306 **EXAMPLES**

33307 None.

33308 **APPLICATION USAGE**

33309 None.

33310 **RATIONALE**

33311 The normal mechanism by which a thread terminates is to return from the routine that was
33312 specified in the *pthread_create*( ) call that started it. The *pthread_exit*( ) function provides the
33313 capability for a thread to terminate without requiring a return from the start routine of that
33314 thread, thereby providing a function analogous to *exit*( ).

33315 Regardless of the method of thread termination, any cancelation cleanup handlers that have
33316 been pushed and not yet popped are executed, and the destructors for any existing thread-
33317 specific data are executed. This volume of IEEE Std 1003.1-200x requires that cancelation
33318 cleanup handlers be popped and called in order. After all cancelation cleanup handlers have
33319 been executed, thread-specific data destructors are called, in an unspecified order, for each item
33320 of thread-specific data that exists in the thread. This ordering is necessary because cancelation

cleanup handlers may rely on thread-specific data.

As the meaning of the status is determined by the application (except when the thread has been canceled, in which case it is PTHREAD_CANCELED), the implementation has no idea what an illegal status value is, which is why no address error checking is done.

FUTURE DIRECTIONS

None.

SEE ALSO

exit( ), *pthread_create*( ), *pthread_join*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_exit*( ) function is marked as part of the Threads option.

33334 **NAME**

33335 pthread_getconcurrency, pthread_setconcurrency — get and set level of concurrency

33336 **SYNOPSIS**

33337 XSI
```
#include <pthread.h>

int pthread_getconcurrency(void);
int pthread_setconcurrency(int new_level);
```
33340

33341 **DESCRIPTION**

33342 Unbound threads in a process may or may not be required to be simultaneously active. By
33343 default, the threads implementation ensures that a sufficient number of threads are active so that
33344 the process can continue to make progress. While this conserves system resources, it may not
33345 produce the most effective level of concurrency.

33346 The *pthread_setconcurrency*( ) function allows an application to inform the threads
33347 implementation of its desired concurrency level, *new_level*. The actual level of concurrency
33348 provided by the implementation as a result of this function call is unspecified.

33349 If *new_level* is zero, it causes the implementation to maintain the concurrency level at its
33350 discretion as if *pthread_setconcurrency*( ) had never been called.

33351 The *pthread_getconcurrency*( ) function shall return the value set by a previous call to the
33352 *pthread_setconcurrency*( ) function. If the *pthread_setconcurrency*( ) function was not previously
33353 called, this function shall return zero to indicate that the implementation is maintaining the
33354 concurrency level.

33355 A call to *pthread_setconcurrency*( ) shall inform the implementation of its desired concurrency |
33356 level. The implementation shall use this as a hint, not a requirement. |

33357 If an implementation does not support multiplexing of user threads on top of several kernel-
33358 scheduled entities, the *pthread_setconcurrency*( ) and *pthread_getconcurrency*( ) functions are |
33359 provided for source code compatibility but they shall have no effect when called. To maintain |
33360 the function semantics, the *new_level* parameter is saved when *pthread_setconcurrency*( ) is called
33361 so that a subsequent call to *pthread_getconcurrency*( ) shall return the same value.

33362 **RETURN VALUE**

33363 If successful, the *pthread_setconcurrency*( ) function shall return zero; otherwise, an error number
33364 shall be returned to indicate the error.

33365 The *pthread_getconcurrency*( ) function shall always return the concurrency level set by a previous
33366 call to *pthread_setconcurrency*( ). If the *pthread_setconcurrency*( ) function has never been called,
33367 *pthread_getconcurrency*( ) shall return zero.

33368 **ERRORS**

33369 The *pthread_setconcurrency*( ) function shall fail if:

33370 [EINVAL] The value specified by *new_level* is negative.

33371 [EAGAIN] The value specific by *new_level* would cause a system resource to be exceeded.

33372 These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Use of these functions changes the state of the underlying concurrency upon which the application depends. Library developers are advised to not use the *pthread_getconcurrency*( ) and *pthread_setconcurrency*( ) functions since their use may conflict with an applications use of these functions.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5.

33388 **NAME**

33389 pthread_getcpuclockid — access a thread CPU-time clock (**ADVANCED REALTIME**
33390 **THREADS**)

33391 **SYNOPSIS**

33392 THR TCT
```
#include <pthread.h>
#include <time.h>

int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);
```
33395

33396 **DESCRIPTION**

33397 The *pthread_getcpuclockid*( ) function shall return in *clock_id* the clock ID of the CPU-time clock of
33398 the thread specified by *thread_id*, if the thread specified by *thread_id* exists.

33399 **RETURN VALUE**

33400 Upon successful completion, *pthread_getcpuclockid*( ) shall return zero; otherwise, an error
33401 number shall be returned to indicate the error.

33402 **ERRORS**

33403 The *pthread_getcpuclockid*( ) function may fail if:

33404 [ESRCH] The value specified by *thread_id* does not refer to an existing thread.

33405 **EXAMPLES**

33406 None.

33407 **APPLICATION USAGE**

33408 The *pthread_getcpuclockid*( ) function is part of the Thread CPU-Time Clocks option and need not
33409 be provided on all implementations.

33410 **RATIONALE**

33411 None.

33412 **FUTURE DIRECTIONS**

33413 None.

33414 **SEE ALSO**

33415 *clock_getcpuclockid*( ), *clock_getres*( ), *timer_create*( ), the Base Definitions volume of
33416 IEEE Std 1003.1-200x, **<pthread.h>**, **<time.h>**

33417 **CHANGE HISTORY**

33418 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

33419 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

pthread_getschedparam, pthread_setschedparam — dynamic thread scheduling parameters access (**REALTIME THREADS**)

SYNOPSIS

THR TPS
```
#include <pthread.h>

int pthread_getschedparam(pthread_t thread, int *restrict policy,
    struct sched_param *restrict param);
int pthread_setschedparam(pthread_t thread, int policy,
    const struct sched_param *param);
```

DESCRIPTION

The *pthread_getschedparam*( ) and *pthread_setschedparam*( ) functions shall, respectively, get and set the scheduling policy and parameters of individual threads within a multi-threaded process to be retrieved and set. For SCHED_FIFO and SCHED_RR, the only required member of the **sched_param** structure is the priority *sched_priority*. For SCHED_OTHER, the affected scheduling parameters are implementation-defined.

The *pthread_getschedparam*( ) function shall retrieve the scheduling policy and scheduling parameters for the thread whose thread ID is given by *thread* and shall store those values in *policy* and *param*, respectively. The priority value returned from *pthread_getschedparam*( ) shall be the value specified by the most recent *pthread_setschedparam*( ), *pthread_setschedprio*( ), or *pthread_create*( ) call affecting the target thread. It shall not reflect any temporary adjustments to its priority as a result of any priority inheritance or ceiling functions. The *pthread_setschedparam*( ) function shall set the scheduling policy and associated scheduling parameters for the thread whose thread ID is given by *thread* to the policy and associated parameters provided in *policy* and *param*, respectively.

The *policy* parameter may have the value SCHED_OTHER, SCHED_FIFO, or SCHED_RR. The scheduling parameters for the SCHED_OTHER policy are implementation-defined. The SCHED_FIFO and SCHED_RR policies shall have a single scheduling parameter, *priority*.

TSP If _POSIX_THREAD_SPORADIC_SERVER is defined, then the *policy* argument may have the value SCHED_SPORADIC, with the exception for the *pthread_setschedparam*( ) function that if the scheduling policy was not SCHED_SPORADIC at the time of the call, it is implementation-defined whether the function is supported; in other words, the implementation need not allow the application to dynamically change the scheduling policy to SCHED_SPORADIC. The sporadic server scheduling policy has the associated parameters *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, *sched_priority*, and *sched_ss_max_repl*. The specified *sched_ss_repl_period* shall be greater than or equal to the specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail. The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the function to succeed; if not, the function shall fail.

If the *pthread_setschedparam*( ) function fails, the scheduling parameters shall not be changed for the target thread.

RETURN VALUE

If successful, the *pthread_getschedparam*( ) and *pthread_setschedparam*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

33464 **ERRORS**

33465 The *pthread_getschedparam*( ) function may fail if:

33466 [ESRCH] The value specified by *thread* does not refer to a existing thread.

33467 The *pthread_setschedparam*( ) function may fail if:

33468 [EINVAL] The value specified by *policy* or one of the scheduling parameters associated
33469 with the scheduling policy *policy* is invalid.

33470 [ENOTSUP] An attempt was made to set the policy or scheduling parameters to an
33471 unsupported value.

33472 TSP [ENOTSUP] An attempt was made to dynamically change the scheduling policy to
33473 SCHED_SPORADIC, and the implementation does not support this change.

33474 [EPERM] The caller does not have the appropriate permission to set either the
33475 scheduling parameters or the scheduling policy of the specified thread.

33476 [EPERM] The implementation does not allow the application to modify one of the
33477 parameters to the value specified.

33478 [ESRCH] The value specified by *thread* does not refer to a existing thread.

33479 These functions shall not return an error code of [EINTR].

33480 **EXAMPLES**

33481 None.

33482 **APPLICATION USAGE**

33483 None.

33484 **RATIONALE**

33485 None.

33486 **FUTURE DIRECTIONS**

33487 None.

33488 **SEE ALSO**

33489 *pthread_setschedprio*( ), *sched_getparam*( ), *sched_getscheduler*( ), the Base Definitions volume of |
33490 IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>**

33491 **CHANGE HISTORY**

33492 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33493 **Issue 6**

33494 The *pthread_getschedparam*( ) and *pthread_setschedparam*( ) functions are marked as part of the
33495 Threads and Thread Execution Scheduling options.

33496 The [ENOSYS] error condition has been removed as stubs need not be provided if an
33497 implementation does not support the Thread Execution Scheduling option.

33498 The Open Group Corrigendum U026/2 is applied, correcting the prototype for the
33499 *pthread_setschedparam*( ) function so that its second argument is of type **int**.

33500 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

33501 The **restrict** keyword is added to the *pthread_getschedparam*( ) prototype for alignment with the
33502 ISO/IEC 9899: 1999 standard.

33503 The Open Group Corrigendum U047/1 is applied. |

IEEE PASC Interpretation 1003.1 #96 is applied, noting that priority values can also be set by a call to the *pthread_setschedprio*( ) function.

33506 **NAME**

33507 pthread_getspecific, pthread_setspecific — thread-specific data management

33508 **SYNOPSIS**

33509 THR
```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *value);
```
33512

33513 **DESCRIPTION**

33514 The *pthread_getspecific*( ) function shall return the value currently bound to the specified *key* on
33515 behalf of the calling thread.

33516 The *pthread_setspecific*( ) function shall associate a thread-specific *value* with a *key* obtained via a
33517 previous call to *pthread_key_create*( ). Different threads may bind different values to the same
33518 key. These values are typically pointers to blocks of dynamically allocated memory that have
33519 been reserved for use by the calling thread.

33520 The effect of calling *pthread_getspecific*( ) or *pthread_setspecific*( ) with a *key* value not obtained
33521 from *pthread_key_create*( ) or after *key* has been deleted with *pthread_key_delete*( ) is undefined.

33522 Both *pthread_getspecific*( ) and *pthread_setspecific*( ) may be called from a thread-specific data
33523 destructor function. A call to *pthread_getspecific*( ) for the thread-specific data key being
33524 destroyed shall return the value NULL, unless the value is changed (after the destructor starts)
33525 by a call to *pthread_setspecific*( ). Calling *pthread_setspecific*( ) from a thread-specific data
33526 destructor routine may result either in lost storage (after at least
33527 PTHREAD_DESTRUCTOR_ITERATIONS attempts at destruction) or in an infinite loop.

33528 Both functions may be implemented as macros.

33529 **RETURN VALUE**

33530 The *pthread_getspecific*( ) function shall return the thread-specific data value associated with the
33531 given *key*. If no thread-specific data value is associated with *key*, then the value NULL shall be
33532 returned.

33533 If successful, the *pthread_setspecific*( ) function shall return zero; otherwise, an error number shall
33534 be returned to indicate the error.

33535 **ERRORS**

33536 No errors are returned from *pthread_getspecific*( ).

33537 The *pthread_setspecific*( ) function shall fail if:

33538 [ENOMEM] Insufficient memory exists to associate the value with the key.

33539 The *pthread_setspecific*( ) function may fail if:

33540 [EINVAL] The key value is invalid.

33541 These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Performance and ease-of-use of *pthread_getspecific*( ) is critical for functions that rely on maintaining state in thread-specific data. Since no errors are required to be detected by it, and since the only error that could be detected is the use of an invalid key, the function to *pthread_getspecific*( ) has been designed to favor speed and simplicity over error reporting.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_key_create( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_getspecific*( ) and *pthread_setspecific*( ) functions are marked as part of the Threads option.

IEEE PASC Interpretation 1003.1c #3 (Part 6) is applied, updating the DESCRIPTION.

33561 **NAME**

33562 pthread_join — wait for thread termination

33563 **SYNOPSIS**

33564 THR
```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```
33565

33566

33567 **DESCRIPTION**

33568 The *pthread_join*( ) function shall suspend execution of the calling thread until the target *thread*
33569 terminates, unless the target *thread* has already terminated. On return from a successful
33570 *pthread_join*( ) call with a non-NULL *value_ptr* argument, the value passed to *pthread_exit*( ) by
33571 the terminating thread shall be made available in the location referenced by *value_ptr*. When a
33572 *pthread_join*( ) returns successfully, the target thread has been terminated. The results of multiple
33573 simultaneous calls to *pthread_join*( ) specifying the same target thread are undefined. If the
33574 thread calling *pthread_join*( ) is canceled, then the target thread shall not be detached.

33575 It is unspecified whether a thread that has exited but remains unjoined counts against
33576 _PTHREAD_THREADS_MAX.

33577 **RETURN VALUE**

33578 If successful, the *pthread_join*( ) function shall return zero; otherwise, an error number shall be
33579 returned to indicate the error.

33580 **ERRORS**

33581 The *pthread_join*( ) function shall fail if:

33582 [EINVAL] The implementation has detected that the value specified by *thread* does not
33583 refer to a joinable thread.

33584 [ESRCH] No thread could be found corresponding to that specified by the given thread
33585 ID.

33586 The *pthread_join*( ) function may fail if:

33587 [EDEADLK] A deadlock was detected or the value of *thread* specifies the calling thread.

33588 The *pthread_join*( ) function shall not return an error code of [EINTR].

33589 **EXAMPLES**

33590 An example of thread creation and deletion follows:

```
33591 typedef struct {
33592     int *ar;
33593     long n;
33594 } subarray;

33595 void *
33596 incer(void *arg)
33597 {
33598     long i;

33599     for (i = 0; i < ((subarray *)arg)->n; i++)
33600         ((subarray *)arg)->ar[i]++;
33601 }

33602 main()
33603 {
33604     int        ar[1000000];
```

```
33605           pthread_t  th1, th2;
33606           subarray   sb1, sb2;

33607           sb1.ar = &ar[0];
33608           sb1.n  = 500000;
33609           (void) pthread_create(&th1, NULL, incer, &sb1);

33610           sb2.ar = &ar[500000];
33611           sb2.n  = 500000;
33612           (void) pthread_create(&th2, NULL, incer, &sb2);

33613           (void) pthread_join(th1, NULL);
33614           (void) pthread_join(th2, NULL);
33615       }
```

33616 **APPLICATION USAGE**

33617 None.

33618 **RATIONALE**

33619 The *pthread_join*( ) function is a convenience that has proven useful in multi-threaded
33620 applications. It is true that a programmer could simulate this function if it were not provided by
33621 passing extra state as part of the argument to the *start_routine*( ). The terminating thread would
33622 set a flag to indicate termination and broadcast a condition that is part of that state; a joining
33623 thread would wait on that condition variable. While such a technique would allow a thread to
33624 wait on more complex conditions (for example, waiting for multiple threads to terminate),
33625 waiting on individual thread termination is considered widely useful. Also, including the
33626 *pthread_join*( ) function in no way precludes a programmer from coding such complex waits.
33627 Thus, while not a primitive, including *pthread_join*( ) in this volume of IEEE Std 1003.1-200x was
33628 considered valuable.

33629 The *pthread_join*( ) function provides a simple mechanism allowing an application to wait for a
33630 thread to terminate. After the thread terminates, the application may then choose to clean up
33631 resources that were used by the thread. For instance, after *pthread_join*( ) returns, any
33632 application-provided stack storage could be reclaimed.

33633 The *pthread_join*( ) or *pthread_detach*( ) function should eventually be called for every thread that
33634 is created with the *detachstate* attribute set to PTHREAD_CREATE_JOINABLE so that storage
33635 associated with the thread may be reclaimed.

33636 The interaction between *pthread_join*( ) and cancelation is well-defined for the following reasons:

33637 • The *pthread_join*( ) function, like all other non-async-cancel-safe functions, can only be called
33638 with deferred cancelability type.

33639 • Cancelation cannot occur in the disabled cancelability state.

33640 Thus, only the default cancelability state need be considered. As specified, either the
33641 *pthread_join*( ) call is canceled, or it succeeds, but not both. The difference is obvious to the
33642 application, since either a cancelation handler is run or *pthread_join*( ) returns. There are no race
33643 conditions since *pthread_join*( ) was called in the deferred cancelability state.

33644 **FUTURE DIRECTIONS**

33645 None.

33646 **SEE ALSO**

33647 *pthread_create*( ), *wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

33648 **CHANGE HISTORY**

33649 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33650 **Issue 6**

33651 The *pthread_join*( ) function is marked as part of the Threads option.

33652 **NAME**

33653 pthread_key_create — thread-specific data key creation

33654 **SYNOPSIS**

33655 THR
```
#include <pthread.h>

int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
```
33656

33657

33658 **DESCRIPTION**

33659 The *pthread_key_create*( ) function shall create a thread-specific data key visible to all threads in
33660 the process. Key values provided by *pthread_key_create*( ) are opaque objects used to locate
33661 thread-specific data. Although the same key value may be used by different threads, the values
33662 bound to the key by *pthread_setspecific*( ) are maintained on a per-thread basis and persist for the
33663 life of the calling thread.

33664 Upon key creation, the value NULL shall be associated with the new key in all active threads.
33665 Upon thread creation, the value NULL shall be associated with all defined keys in the new
33666 thread.

33667 An optional destructor function may be associated with each key value. At thread exit, if a key
33668 value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with
33669 that key, the value of the key is set to NULL, and then the function pointed to is called with the
33670 previously associated value as its sole argument. The order of destructor calls is unspecified if
33671 more than one destructor exists for a thread when it exits.

33672 If, after all the destructors have been called for all non-NULL values with associated destructors,
33673 there are still some non-NULL values with associated destructors, then the process is repeated.
33674 If, after at least {PTHREAD_DESTRUCTOR_ITERATIONS} iterations of destructor calls for
33675 outstanding non-NULL values, there are still some non-NULL values with associated
33676 destructors, implementations may stop calling destructors, or they may continue calling
33677 destructors until no non-NULL values with associated destructors exist, even though this might
33678 result in an infinite loop.

33679 **RETURN VALUE**

33680 If successful, the *pthread_key_create*( ) function shall store the newly created key value at **key* and
33681 shall return zero. Otherwise, an error number shall be returned to indicate the error.

33682 **ERRORS**

33683 The *pthread_key_create*( ) function shall fail if:

33684 [EAGAIN] The system lacked the necessary resources to create another thread-specific
33685 data key, or the system-imposed limit on the total number of keys per process
33686 PTHREAD_KEYS_MAX has been exceeded.

33687 [ENOMEM] Insufficient memory exists to create the key.

33688 The *pthread_key_create*( ) function shall not return an error code of [EINTR].

EXAMPLES

The following example demonstrates a function that initializes a thread-specific data key when it is first called, and associates a thread-specific object with each calling thread, initializing this object when necessary.

```
static pthread_key_t key;
static pthread_once_t key_once = PTHREAD_ONCE_INIT;

static void
make_key()
{
    (void) pthread_key_create(&key, NULL);
}

func()
{
    void *ptr;

    (void) pthread_once(&key_once, make_key);
    if ((ptr = pthread_getspecific(key)) == NULL) {
        ptr = malloc(OBJECT_SIZE);
        ...
        (void) pthread_setspecific(key, ptr);
    }
    ...
}
```

Note that the key has to be initialized before *pthread_getspecific*() or *pthread_setspecific*() can be used. The *pthread_key_create*() call could either be explicitly made in a module initialization routine, or it can be done implicitly by the first call to a module as in this example. Any attempt to use the key before it is initialized is a programming error, making the code below incorrect.

```
static pthread_key_t key;

func()
{
    void *ptr;

   /* KEY NOT INITIALIZED!!!  THIS WON'T WORK!!! */
    if ((ptr = pthread_getspecific(key)) == NULL &&
        pthread_setspecific(key, NULL) != 0) {
        pthread_key_create(&key, NULL);
        ...
    }
}
```

APPLICATION USAGE

None.

RATIONALE

Destructor Functions

Normally, the value bound to a key on behalf of a particular thread is a pointer to storage allocated dynamically on behalf of the calling thread. The destructor functions specified with *pthread_key_create*( ) are intended to be used to free this storage when the thread exits. Thread cancelation cleanup handlers cannot be used for this purpose because thread-specific data may persist outside the lexical scope in which the cancelation cleanup handlers operate.

If the value associated with a key needs to be updated during the lifetime of the thread, it may be necessary to release the storage associated with the old value before the new value is bound. Although the *pthread_setspecific*( ) function could do this automatically, this feature is not needed often enough to justify the added complexity. Instead, the programmer is responsible for freeing the stale storage:

```
pthread_getspecific(key, &old);
new = allocate();
destructor(old);
pthread_setspecific(key, new);
```

Note: The above example could leak storage if run with asynchronous cancelation enabled. No such problems occur in the default cancelation state if no cancelation points occur between the get and set.

There is no notion of a destructor-safe function. If an application does not call *pthread_exit*( ) from a signal handler, or if it blocks any signal whose handler may call *pthread_exit*( ) while calling async-unsafe functions, all functions may be safely called from destructors.

Non-Idempotent Data Key Creation

There were requests to make *pthread_key_create*( ) idempotent with respect to a given *key* address parameter. This would allow applications to call *pthread_key_create*( ) multiple times for a given *key* address and be guaranteed that only one key would be created. Doing so would require the key value to be previously initialized (possibly at compile time) to a known null value and would require that implicit mutual-exclusion be performed based on the address and contents of the *key* parameter in order to guarantee that exactly one key would be created.

Unfortunately, the implicit mutual-exclusion would not be limited to only *pthread_key_create*( ). On many implementations, implicit mutual-exclusion would also have to be performed by *pthread_getspecific*( ) and *pthread_setspecific*( ) in order to guard against using incompletely stored or not-yet-visible key values. This could significantly increase the cost of important operations, particularly *pthread_getspecific*( ).

Thus, this proposal was rejected. The *pthread_key_create*( ) function performs no implicit synchronization. It it the responsibility of the programmer to ensure that it is called exactly once per key before use of the key. Several straightforward mechanisms can already be used to accomplish this, including calling explicit module initialization functions, using mutexes, and using *pthread_once*( ). This places no significant burden on the programmer, introduces no possibly confusing *ad hoc* implicit synchronization mechanism, and potentially allows commonly used thread-specific data operations to be more efficient.

FUTURE DIRECTIONS

None.

33771 **SEE ALSO**

33772 *pthread_getspecific*( ), *pthread_key_delete*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
33773 **<pthread.h>**

33774 **CHANGE HISTORY**

33775 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33776 **Issue 6**

33777 The *pthread_key_create*( ) function is marked as part of the Threads option.

33778 IEEE PASC Interpretation 1003.1c #8 is applied, updating the DESCRIPTION.

NAME

pthread_key_delete — thread-specific data key deletion

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_key_delete(pthread_key_t key);
```

DESCRIPTION

The *pthread_key_delete*( ) function shall delete a thread-specific data key previously returned by *pthread_key_create*( ). The thread-specific data values associated with *key* need not be NULL at the time *pthread_key_delete*( ) is called. It is the responsibility of the application to free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads; this cleanup can be done either before or after *pthread_key_delete*( ) is called. Any attempt to use *key* following the call to *pthread_key_delete*( ) results in undefined behavior.

The *pthread_key_delete*( ) function shall be callable from within destructor functions. No destructor functions shall be invoked by *pthread_key_delete*( ). Any destructor function that may have been associated with *key* shall no longer be called upon thread exit.

RETURN VALUE

If successful, the *pthread_key_delete*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_key_delete*( ) function may fail if:

[EINVAL] The *key* value is invalid.

The *pthread_key_delete*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

A thread-specific data key deletion function has been included in order to allow the resources associated with an unused thread-specific data key to be freed. Unused thread-specific data keys can arise, among other scenarios, when a dynamically loaded module that allocated a key is unloaded.

Conforming applications are responsible for performing any cleanup actions needed for data structures associated with the key to be deleted, including data referenced by thread-specific data values. No such cleanup is done by *pthread_key_delete*( ). In particular, destructor functions are not called. There are several reasons for this division of responsibility:

1. The associated destructor functions used to free thread-specific data at thread exit time are only guaranteed to work correctly when called in the thread that allocated the thread-specific data. (Destructors themselves may utilize thread-specific data.) Thus, they cannot be used to free thread-specific data in other threads at key deletion time. Attempting to have them called by other threads at key deletion time would require other threads to be asynchronously interrupted. But since interrupted threads could be in an arbitrary state, including holding locks necessary for the destructor to run, this approach would fail. In general, there is no safe mechanism whereby an implementation could free thread-specific data at key deletion time.

2. Even if there were a means of safely freeing thread-specific data associated with keys to be deleted, doing so would require that implementations be able to enumerate the threads with non-NULL data and potentially keep them from creating more thread-specific data while the key deletion is occurring. This special case could cause extra synchronization in the normal case, which would otherwise be unnecessary.

For an application to know that it is safe to delete a key, it has to know that all the threads that might potentially ever use the key do not attempt to use it again. For example, it could know this if all the client threads have called a cleanup procedure declaring that they are through with the module that is being shut down, perhaps by zero'ing a reference count.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_key_create( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_key_delete*( ) function is marked as part of the Threads option.

33842 **NAME**

33843 pthread_kill — send a signal to a thread

33844 **SYNOPSIS**

33845 THR
```
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);
```
33846

33847

33848 **DESCRIPTION**

33849 The *pthread_kill*( ) function shall request that a signal be delivered to the specified thread. |

33850 As in *kill*( ), if *sig* is zero, error checking shall be performed but no signal shall actually be sent. |

33851 **RETURN VALUE**

33852 Upon successful completion, the function shall return a value of zero. Otherwise, the function
33853 shall return an error number. If the *pthread_kill*( ) function fails, no signal shall be sent.

33854 **ERRORS**

33855 The *pthread_kill*( ) function shall fail if:

33856 [ESRCH] No thread could be found corresponding to that specified by the given thread
33857 ID.

33858 [EINVAL] The value of the *sig* argument is an invalid or unsupported signal number.

33859 The *pthread_kill*( ) function shall not return an error code of [EINTR].

33860 **EXAMPLES**

33861 None.

33862 **APPLICATION USAGE**

33863 The *pthread_kill*( ) function provides a mechanism for asynchronously directing a signal at a
33864 thread in the calling process. This could be used, for example, by one thread to affect broadcast
33865 delivery of a signal to a set of threads.

33866 Note that *pthread_kill*( ) only causes the signal to be handled in the context of the given thread;
33867 the signal action (termination or stopping) affects the process as a whole.

33868 **RATIONALE**

33869 None.

33870 **FUTURE DIRECTIONS**

33871 None.

33872 **SEE ALSO**

33873 *kill*( ), *pthread_self*( ), *raise*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

33874 **CHANGE HISTORY**

33875 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33876 **Issue 6**

33877 The *pthread_kill*( ) function is marked as part of the Threads option.

33878 The APPLICATION USAGE section is added.

NAME

pthread_mutex_destroy, pthread_mutex_init — destroy and initialize a mutex

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

DESCRIPTION

The *pthread_mutex_destroy*( ) function shall destroy the mutex object referenced by *mutex*; the mutex object becomes, in effect, uninitialized. An implementation may cause *pthread_mutex_destroy*( ) to set the object referenced by *mutex* to an invalid value. A destroyed mutex object can be reinitialized using *pthread_mutex_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

The *pthread_mutex_init*( ) function shall initialize the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Only *mutex* itself may be used for performing synchronization. The result of referring to copies of *mutex* in calls to *pthread_mutex_lock*( ), *pthread_mutex_trylock*( ), *pthread_mutex_unlock*( ), and *pthread_mutex_destroy*( ) is undefined.

Attempting to initialize an already initialized mutex results in undefined behavior.

In cases where default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes that are statically allocated. The effect shall be equivalent to dynamic initialization by a call to *pthread_mutex_init*( ) with parameter *attr* specified as NULL, except that no error checks are performed.

RETURN VALUE

If successful, the *pthread_mutex_destroy*( ) and *pthread_mutex_init*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

The [EBUSY] and [EINVAL] error checks, if implemented, act as if they were performed immediately at the beginning of processing for the function and shall cause an error return prior to modifying the state of the mutex specified by *mutex*.

ERRORS

The *pthread_mutex_destroy*( ) function may fail if:

[EBUSY] The implementation has detected an attempt to destroy the object referenced by *mutex* while it is locked or referenced (for example, while being used in a *pthread_cond_timedwait*( ) or *pthread_cond_wait*( )) by another thread.

[EINVAL] The value specified by *mutex* is invalid.

The *pthread_mutex_init*( ) function shall fail if:

[EAGAIN] The system lacked the necessary resources (other than memory) to initialize another mutex.

[ENOMEM] Insufficient memory exists to initialize the mutex.

[EPERM] The caller does not have the privilege to perform the operation.

The *pthread_mutex_init*( ) function may fail if:

[EBUSY] The implementation has detected an attempt to reinitialize the object referenced by *mutex*, a previously initialized, but not yet destroyed, mutex. |

[EINVAL] The value specified by *attr* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Alternate Implementations Possible

This volume of IEEE Std 1003.1-200x supports several alternative implementations of mutexes. An implementation may store the lock directly in the object of type **pthread_mutex_t**. Alternatively, an implementation may store the lock in the heap and merely store a pointer, handle, or unique ID in the mutex object. Either implementation has advantages or may be required on certain hardware configurations. So that portable code can be written that is invariant to this choice, this volume of IEEE Std 1003.1-200x does not define assignment or equality for this type, and it uses the term ‘‘initialize’’ to reinforce the (more restrictive) notion that the lock may actually reside in the mutex object itself.

Note that this precludes an over-specification of the type of the mutex or condition variable and motivates the opacity of the type.

An implementation is permitted, but not required, to have *pthread_mutex_destroy*( ) store an illegal value into the mutex. This may help detect erroneous programs that try to lock (or otherwise reference) a mutex that has already been destroyed.

Tradeoff Between Error Checks and Performance Supported

Many of the error checks were made optional in order to let implementations trade off performance *versus* degree of error checking according to the needs of their specific applications and execution environment. As a general rule, errors or conditions caused by the system (such as insufficient memory) always need to be reported, but errors due to an erroneously coded application (such as failing to provide adequate synchronization to prevent a mutex from being deleted while in use) are made optional.

A wide range of implementations is thus made possible. For example, an implementation intended for application debugging may implement all of the error checks, but an implementation running a single, provably correct application under very tight performance constraints in an embedded computer might implement minimal checks. An implementation might even be provided in two versions, similar to the options that compilers provide: a full-checking, but slower version; and a limited-checking, but faster version. To forbid this optionality would be a disservice to users.

By carefully limiting the use of ‘‘undefined behavior’’ only to things that an erroneous (badly coded) application might do, and by defining that resource-not-available errors are mandatory, this volume of IEEE Std 1003.1-200x ensures that a fully-conforming application is portable

across the full range of implementations, while not forcing all implementations to add overhead to check for numerous things that a correct program never does.

Why No Limits Defined

Defining symbols for the maximum number of mutexes and condition variables was considered but rejected because the number of these objects may change dynamically. Furthermore, many implementations place these objects into application memory; thus, there is no explicit maximum.

Static Initializers for Mutexes and Condition Variables

Providing for static initialization of statically allocated synchronization objects allows modules with private static synchronization variables to avoid runtime initialization tests and overhead. Furthermore, it simplifies the coding of self-initializing modules. Such modules are common in C libraries, where for various reasons the design calls for self-initialization instead of requiring an explicit module initialization function to be called. An example use of static initialization follows.

Without static initialization, a self-initializing routine *foo*( ) might look as follows:

```
static pthread_once_t foo_once = PTHREAD_ONCE_INIT;
static pthread_mutex_t foo_mutex;

void foo_init()
{
    pthread_mutex_init(&foo_mutex, NULL);
}

void foo()
{
    pthread_once(&foo_once, foo_init);
    pthread_mutex_lock(&foo_mutex);
   /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

With static initialization, the same routine could be coded as follows:

```
static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;

void foo()
{
    pthread_mutex_lock(&foo_mutex);
   /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

Note that the static initialization both eliminates the need for the initialization test inside *pthread_once*( ) and the fetch of &*foo_mutex* to learn the address to be passed to *pthread_mutex_lock*( ) or *pthread_mutex_unlock*( ).

Thus, the C code written to initialize static objects is simpler on all systems and is also faster on a large class of systems; those where the (entire) synchronization object can be stored in application memory.

Yet the locking performance question is likely to be raised for machines that require mutexes to be allocated out of special memory. Such machines actually have to have mutexes and possibly

34010 condition variables contain pointers to the actual hardware locks. For static initialization to work
34011 on such machines, *pthread_mutex_lock*( ) also has to test whether or not the pointer to the actual
34012 lock has been allocated. If it has not, *pthread_mutex_lock*( ) has to initialize it before use. The
34013 reservation of such resources can be made when the program is loaded, and hence return codes
34014 have not been added to mutex locking and condition variable waiting to indicate failure to
34015 complete initialization.

34016 This runtime test in *pthread_mutex_lock*( ) would at first seem to be extra work; an extra test is
34017 required to see whether the pointer has been initialized. On most machines this would actually
34018 be implemented as a fetch of the pointer, testing the pointer against zero, and then using the
34019 pointer if it has already been initialized. While the test might seem to add extra work, the extra
34020 effort of testing a register is usually negligible since no extra memory references are actually
34021 done. As more and more machines provide caches, the real expenses are memory references, not
34022 instructions executed.

34023 Alternatively, depending on the machine architecture, there are often ways to eliminate *all*
34024 overhead in the most important case: on the lock operations that occur *after* the lock has been
34025 initialized. This can be done by shifting more overhead to the less frequent operation:
34026 initialization. Since out-of-line mutex allocation also means that an address has to be
34027 dereferenced to find the actual lock, one technique that is widely applicable is to have static
34028 initialization store a bogus value for that address; in particular, an address that causes a machine
34029 fault to occur. When such a fault occurs upon the first attempt to lock such a mutex, validity
34030 checks can be done, and then the correct address for the actual lock can be filled in. Subsequent
34031 lock operations incur no extra overhead since they do not ‘‘fault’’. This is merely one technique
34032 that can be used to support static initialization, while not adversely affecting the performance of
34033 lock acquisition. No doubt there are other techniques that are highly machine-dependent.

34034 The locking overhead for machines doing out-of-line mutex allocation is thus similar for
34035 modules being implicitly initialized, where it is improved for those doing mutex allocation
34036 entirely inline. The inline case is thus made much faster, and the out-of-line case is not
34037 significantly worse.

34038 Besides the issue of locking performance for such machines, a concern is raised that it is possible
34039 that threads would serialize contending for initialization locks when attempting to finish
34040 initializing statically allocated mutexes. (Such finishing would typically involve taking an
34041 internal lock, allocating a structure, storing a pointer to the structure in the mutex, and releasing
34042 the internal lock.) First, many implementations would reduce such serialization by hashing on
34043 the mutex address. Second, such serialization can only occur a bounded number of times. In
34044 particular, it can happen at most as many times as there are statically allocated synchronization
34045 objects. Dynamically allocated objects would still be initialized via *pthread_mutex_init*( ) or
34046 *pthread_cond_init*( ).

34047 Finally, if none of the above optimization techniques for out-of-line allocation yields sufficient
34048 performance for an application on some implementation, the application can avoid static
34049 initialization altogether by explicitly initializing all synchronization objects with the
34050 corresponding *pthread_*_init*( ) functions, which are supported by all implementations. An
34051 implementation can also document the tradeoffs and advise which initialization technique is
34052 more efficient for that particular implementation.

Destroying Mutexes

A mutex can be destroyed immediately after it is unlocked. For example, consider the following code:

```
struct obj {
pthread_mutex_t om;
    int refcnt;
    ...
};

obj_done(struct obj *op)
{
    pthread_mutex_lock(&op->om);
    if (--op->refcnt == 0) {
        pthread_mutex_unlock(&op->om);
(A)     pthread_mutex_destroy(&op->om);
(B)     free(op);
    } else
(C)     pthread_mutex_unlock(&op->om);
}
```

In this case *obj* is reference counted and *obj_done*( ) is called whenever a reference to the object is dropped. Implementations are required to allow an object to be destroyed and freed and potentially unmapped (for example, lines A and B) immediately after the object is unlocked (line C).

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_getprioceiling( ), *pthread_mutex_lock*( ), *pthread_mutex_timedlock*( ), *pthread_mutexattr_getpshared*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_mutex_destroy*( ) and *pthread_mutex_init*( ) functions are marked as part of the Threads option.

The *pthread_mutex_timedlock*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

The **restrict** keyword is added to the *pthread_mutex_init*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

pthread_mutex_getprioceiling, pthread_mutex_setprioceiling — get and set the priority ceiling of a mutex (**REALTIME THREADS**)

SYNOPSIS

THR TPP
```
#include <pthread.h>

int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
    int *restrict prioceiling);
int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
    int prioceiling, int *restrict old_ceiling);
```

DESCRIPTION

The *pthread_mutex_getprioceiling*( ) function shall return the current priority ceiling of the mutex.

The *pthread_mutex_setprioceiling*( ) function shall either lock the mutex if it is unlocked, or block until it can successfully lock the mutex, then it shall change the mutex's priority ceiling and release the mutex. When the change is successful, the previous value of the priority ceiling shall be returned in *old_ceiling*. The process of locking the mutex need not adhere to the priority protect protocol.

If the *pthread_mutex_setprioceiling*( ) function fails, the mutex priority ceiling shall not be changed.

RETURN VALUE

If successful, the *pthread_mutex_getprioceiling*( ) and *pthread_mutex_setprioceiling*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutex_getprioceiling*( ) and *pthread_mutex_setprioceiling*( ) functions may fail if:

[EINVAL] The priority requested by *prioceiling* is out of range.

[EINVAL] The value specified by *mutex* does not refer to a currently existing mutex.

[EPERM] The caller does not have the privilege to perform the operation.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_destroy( ), *pthread_mutex_lock*( ), *pthread_mutex_timedlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_mutex_getprioceiling*( ) and *pthread_mutex_setprioceiling*( ) functions are marked as part of the Threads and Thread Priority Protection options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Priority Protection option.

The [ENOSYS] error denoting non-support of the priority ceiling protocol for mutexes has been removed. This is since if the implementation provides the functions (regardless of whether _POSIX_PTHREAD_PRIO_PROTECT is defined), they must function as in the DESCRIPTION and therefore the priority ceiling protocol for mutexes is supported.

The *pthread_mutex_timedlock*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *pthread_mutex_getprioceiling*( ) and *pthread_mutex_setprioceiling*( ) prototypes for alignment with the ISO / IEC 9899: 1999 standard.

34146 **NAME**

34147 pthread_mutex_init — initialize a mutex

34148 **SYNOPSIS**

34149 THR
```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```
34153

34154 **DESCRIPTION**

34155 Refer to *pthread_mutex_destroy*( ).

NAME

pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a mutex

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

The mutex object referenced by *mutex* shall be locked by calling *pthread_mutex_lock*(). If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner.

XSI If the mutex type is PTHREAD_MUTEX_NORMAL, deadlock detection shall not be provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results.

If the mutex type is PTHREAD_MUTEX_ERRORCHECK, then error checking shall be provided. If a thread attempts to relock a mutex that it has already locked, an error shall be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error shall be returned.

If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex shall maintain the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count shall be set to one. Every time a thread relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks the mutex, the lock count shall be decremented by one. When the lock count reaches zero, the mutex shall become available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error shall be returned.

If the mutex type is PTHREAD_MUTEX_DEFAULT, attempting to recursively lock the mutex results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in undefined behavior.

The *pthread_mutex_trylock*() function shall be equivalent to *pthread_mutex_lock*(), except that if the mutex object referenced by *mutex* is currently locked (by any thread, including the current thread), the call shall return immediately. If the mutex type is PTHREAD_MUTEX_RECURSIVE and the mutex is currently owned by the calling thread, the mutex lock count shall be incremented by one and the *pthread_mutex_trylock*() function shall immediately return success.

XSI The *pthread_mutex_unlock*() function shall release the mutex object referenced by *mutex*. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock*() is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

XSI (In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the mutex shall become available when the count reaches zero and the calling thread no longer has any locks on this mutex).

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread shall resume waiting for the mutex as if it was not interrupted.

34202 **RETURN VALUE**

34203 If successful, the *pthread_mutex_lock*( ) and *pthread_mutex_unlock*( ) functions shall return zero;
34204 otherwise, an error number shall be returned to indicate the error.

34205 The *pthread_mutex_trylock*( ) function shall return zero if a lock on the mutex object referenced by
34206 *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

34207 **ERRORS**

34208 The *pthread_mutex_lock*( ) and *pthread_mutex_trylock*( ) functions shall fail if:

34209 [EINVAL] The *mutex* was created with the protocol attribute having the value
34210 PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than
34211 the mutex's current priority ceiling.

34212 The *pthread_mutex_trylock*( ) function shall fail if:

34213 [EBUSY] The *mutex* could not be acquired because it was already locked.

34214 The *pthread_mutex_lock*( ), *pthread_mutex_trylock*( ), and *pthread_mutex_unlock*( ) functions may
34215 fail if:

34216 [EINVAL] The value specified by *mutex* does not refer to an initialized mutex object.

34217 XSI [EAGAIN] The mutex could not be acquired because the maximum number of recursive
34218 locks for *mutex* has been exceeded.

34219 The *pthread_mutex_lock*( ) function may fail if:

34220 [EDEADLK] The current thread already owns the mutex.

34221 The *pthread_mutex_unlock*( ) function may fail if:

34222 [EPERM] The current thread does not own the mutex.

34223 These functions shall not return an error code of [EINTR].

34224 **EXAMPLES**

34225 None.

34226 **APPLICATION USAGE**

34227 None.

34228 **RATIONALE**

34229 Mutex objects are intended to serve as a low-level primitive from which other thread
34230 synchronization functions can be built. As such, the implementation of mutexes should be as
34231 efficient as possible, and this has ramifications on the features available at the interface.

34232 The mutex functions and the particular default settings of the mutex attributes have been
34233 motivated by the desire to not preclude fast, inlined implementations of mutex locking and
34234 unlocking.

34235 For example, deadlocking on a double-lock is explicitly allowed behavior in order to avoid
34236 requiring more overhead in the basic mechanism than is absolutely necessary. (More ''friendly''
34237 mutexes that detect deadlock or that allow multiple locking by the same thread are easily
34238 constructed by the user via the other mechanisms provided. For example, *pthread_self*( ) can be
34239 used to record mutex ownership.) Implementations might also choose to provide such extended
34240 features as options via special mutex attributes.

34241 Since most attributes only need to be checked when a thread is going to be blocked, the use of
34242 attributes does not slow the (common) mutex-locking case.

Likewise, while being able to extract the thread ID of the owner of a mutex might be desirable, it would require storing the current thread ID when each mutex is locked, and this could incur unacceptable levels of overhead. Similar arguments apply to a *mutex_tryunlock* operation.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_destroy( ), *pthread_mutex_timedlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_mutex_lock*( ), *pthread_mutex_trylock*( ), and *pthread_mutex_unlock*( ) functions are marked as part of the Threads option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The behavior when attempting to relock a mutex is defined.

The *pthread_mutex_timedlock*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

34261 **NAME**

34262 pthread_mutex_setprioceiling — change the priority ceiling of a mutex (**REALTIME**
34263 **THREADS**)

34264 **SYNOPSIS**

34265 THR TPP
```
#include <pthread.h>
```

34266
```
int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
    int prioceiling, int *restrict old_ceiling);
```
34267
34268

34269 **DESCRIPTION**

34270 Refer to *pthread_mutex_getprioceiling*( ).

NAME

pthread_mutex_timedlock — lock a mutex (**ADVANCED REALTIME**)

SYNOPSIS

THR TMO
```
#include <pthread.h>
#include <time.h>

int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abs_timeout);
```

DESCRIPTION

The *pthread_mutex_timedlock*( ) function shall lock the mutex object referenced by *mutex*. If the mutex is already locked, the calling thread shall block until the mutex becomes available as in the *pthread_mutex_lock*( ) function. If the mutex cannot be locked without waiting for another thread to unlock the mutex, this wait shall be terminated when the specified timeout expires.

The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout shall be based on the system clock as returned by the *time*( ) function.

The resolution of the timeout shall be the resolution of the clock on which it is based. The **timespec** data type is defined in the **<time.h>** header.

Under no circumstance shall the function fail with a timeout if the mutex can be locked immediately. The validity of the *abs_timeout* parameter need not be checked if the mutex can be locked immediately.

As a consequence of the priority inheritance rules (for mutexes initialized with the PRIO_INHERIT protocol), if a timed mutex wait is terminated because its timeout expires, the priority of the owner of the mutex shall be adjusted as necessary to reflect the fact that this thread is no longer among the threads waiting for the mutex.

RETURN VALUE

If successful, the *pthread_mutex_timedlock*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutex_timedlock*( ) function shall fail if:

[EINVAL] The mutex was created with the protocol attribute having the value PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than the mutex' current priority ceiling.

[EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1 000 million.

[ETIMEDOUT] The mutex could not be locked before the specified timeout expired.

The *pthread_mutex_timedlock*( ) function may fail if:

[EINVAL] The value specified by mutex does not refer to an initialized mutex object.

XSI [EAGAIN] The mutex could not be acquired because the maximum number of recursive locks for mutex has been exceeded.

[EDEADLK] The current thread already owns the mutex.

This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_mutex_timedlock*() function is part of the Threads and Timeouts options and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_destroy(), *pthread_mutex_lock*(), *pthread_mutex_trylock*(), *time*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

34332 **NAME**

34333 pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a mutex

34334 **SYNOPSIS**

34335 THR
```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```
34336

34337

34338

34339 **DESCRIPTION**

34340 Refer to *pthread_mutex_lock*( ).

34341 **NAME**

34342 pthread_mutexattr_destroy, pthread_mutexattr_init — destroy and initialize mutex attributes
34343 object

34344 **SYNOPSIS**

34345 THR
```
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```
34348

34349 **DESCRIPTION**

34350 The *pthread_mutexattr_destroy*( ) function shall destroy a mutex attributes object; the object
34351 becomes, in effect, uninitialized. An implementation may cause *pthread_mutexattr_destroy*( ) to
34352 set the object referenced by *attr* to an invalid value. A destroyed *attr* attributes object can be |
34353 reinitialized using *pthread_mutexattr_init*( ); the results of otherwise referencing the object after it |
34354 has been destroyed are undefined. |

34355 The *pthread_mutexattr_init*( ) function shall initialize a mutex attributes object *attr* with the
34356 default value for all of the attributes defined by the implementation.

34357 Results are undefined if *pthread_mutexattr_init*( ) is called specifying an already initialized *attr* |
34358 attributes object. |

34359 After a mutex attributes object has been used to initialize one or more mutexes, any function
34360 affecting the attributes object (including destruction) shall not affect any previously initialized |
34361 mutexes. |

34362 **RETURN VALUE**

34363 Upon successful completion, *pthread_mutexattr_destroy*( ) and *pthread_mutexattr_init*( ) shall
34364 return zero; otherwise, an error number shall be returned to indicate the error.

34365 **ERRORS**

34366 The *pthread_mutexattr_destroy*( ) function may fail if:

34367 [EINVAL] The value specified by *attr* is invalid.

34368 The *pthread_mutexattr_init*( ) function shall fail if:

34369 [ENOMEM] Insufficient memory exists to initialize the mutex attributes object.

34370 These functions shall not return an error code of [EINTR].

34371 **EXAMPLES**

34372 None.

34373 **APPLICATION USAGE**

34374 None.

34375 **RATIONALE**

34376 See *pthread_attr_init*( ) for a general explanation of attributes. Attributes objects allow
34377 implementations to experiment with useful extensions and permit extension of this volume of
34378 IEEE Std 1003.1-200x without changing the existing functions. Thus, they provide for future
34379 extensibility of this volume of IEEE Std 1003.1-200x and reduce the temptation to standardize
34380 prematurely on semantics that are not yet widely implemented or understood.

34381 Examples of possible additional mutex attributes that have been discussed are *spin_only*,
34382 *limited_spin*, *no_spin*, *recursive*, and *metered*. (To explain what the latter attributes might mean:
34383 recursive mutexes would allow for multiple re-locking by the current owner; metered mutexes
34384 would transparently keep records of queue length, wait time, and so on.) Since there is not yet

wide agreement on the usefulness of these resulting from shared implementation and usage experience, they are not yet specified in this volume of IEEE Std 1003.1-200x. Mutex attributes objects, however, make it possible to test out these concepts for possible standardization at a later time.

Mutex Attributes and Performance

Care has been taken to ensure that the default values of the mutex attributes have been defined such that mutexes initialized with the defaults have simple enough semantics so that the locking and unlocking can be done with the equivalent of a test-and-set instruction (plus possibly a few other basic instructions).

There is at least one implementation method that can be used to reduce the cost of testing at lock-time if a mutex has non-default attributes. One such method that an implementation can employ (and this can be made fully transparent to fully conforming POSIX applications) is to secretly pre-lock any mutexes that are initialized to non-default attributes. Any later attempt to lock such a mutex causes the implementation to branch to the ''slow path'' as if the mutex were unavailable; then, on the slow path, the implementation can do the ''real work'' to lock a non-default mutex. The underlying unlock operation is more complicated since the implementation never really wants to release the pre-lock on this kind of mutex. This illustrates that, depending on the hardware, there may be certain optimizations that can be used so that whatever mutex attributes are considered ''most frequently used'' can be processed most efficiently.

Process Shared Memory and Synchronization

The existence of memory mapping functions in this volume of IEEE Std 1003.1-200x leads to the possibility that an application may allocate the synchronization objects from this section in memory that is accessed by multiple processes (and therefore, by threads of multiple processes).

In order to permit such usage, while at the same time keeping the usual case (that is, usage within a single process) efficient, a process-shared option has been defined.

If an implementation supports the _POSIX_THREAD_PROCESS_SHARED option, then the *process-shared* attribute can be used to indicate that mutexes or condition variables may be accessed by threads of multiple processes.

The default setting of PTHREAD_PROCESS_PRIVATE has been chosen for the *process-shared* attribute so that the most efficient forms of these synchronization objects are created by default.

Synchronization variables that are initialized with the PTHREAD_PROCESS_PRIVATE *process-shared* attribute may only be operated on by threads in the process that initialized them. Synchronization variables that are initialized with the PTHREAD_PROCESS_SHARED *process-shared* attribute may be operated on by any thread in any process that has access to it. In particular, these processes may exist beyond the lifetime of the initializing process. For example, the following code implements a simple counting semaphore in a mapped file that may be used by many processes.

```
/* sem.h */
struct semaphore {
    pthread_mutex_t lock;
    pthread_cond_t nonzero;
    unsigned count;
};
typedef struct semaphore semaphore_t;

semaphore_t *semaphore_create(char *semaphore_name);
semaphore_t *semaphore_open(char *semaphore_name);
```

```
34431       void semaphore_post(semaphore_t *semap);
34432       void semaphore_wait(semaphore_t *semap);
34433       void semaphore_close(semaphore_t *semap);

34434       /* sem.c */
34435       #include <sys/types.h>
34436       #include <sys/stat.h>
34437       #include <sys/mman.h>
34438       #include <fcntl.h>
34439       #include <pthread.h>
34440       #include "sem.h"

34441       semaphore_t *
34442       semaphore_create(char *semaphore_name)
34443       {
34444       int fd;
34445           semaphore_t *semap;
34446           pthread_mutexattr_t psharedm;
34447           pthread_condattr_t psharedc;

34448           fd = open(semaphore_name, O_RDWR | O_CREAT | O_EXCL, 0666);
34449           if (fd < 0)
34450               return (NULL);
34451           (void) ftruncate(fd, sizeof(semaphore_t));
34452           (void) pthread_mutexattr_init(&psharedm);
34453           (void) pthread_mutexattr_setpshared(&psharedm,
34454               PTHREAD_PROCESS_SHARED);
34455           (void) pthread_condattr_init(&psharedc);
34456           (void) pthread_condattr_setpshared(&psharedc,
34457               PTHREAD_PROCESS_SHARED);
34458           semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
34459                   PROT_READ | PROT_WRITE, MAP_SHARED,
34460                   fd, 0);
34461           close (fd);
34462           (void) pthread_mutex_init(&semap->lock, &psharedm);
34463           (void) pthread_cond_init(&semap->nonzero, &psharedc);
34464           semap->count = 0;
34465           return (semap);
34466       }

34467       semaphore_t *
34468       semaphore_open(char *semaphore_name)
34469       {
34470           int fd;
34471           semaphore_t *semap;

34472           fd = open(semaphore_name, O_RDWR, 0666);
34473           if (fd < 0)
34474               return (NULL);
34475           semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
34476                   PROT_READ | PROT_WRITE, MAP_SHARED,
34477                   fd, 0);
34478           close (fd);
34479           return (semap);
34480       }
```

```
34481       void
34482       semaphore_post(semaphore_t *semap)
34483       {
34484           pthread_mutex_lock(&semap->lock);
34485           if (semap->count == 0)
34486               pthread_cond_signal(&semapx->nonzero);
34487           semap->count++;
34488           pthread_mutex_unlock(&semap->lock);
34489       }

34490       void
34491       semaphore_wait(semaphore_t *semap)
34492       {
34493           pthread_mutex_lock(&semap->lock);
34494           while (semap->count == 0)
34495               pthread_cond_wait(&semap->nonzero, &semap->lock);
34496           semap->count--;
34497           pthread_mutex_unlock(&semap->lock);
34498       }

34499       void
34500       semaphore_close(semaphore_t *semap)
34501       {
34502           munmap((void *) semap, sizeof(semaphore_t));
34503       }
```

34504 The following code is for three separate processes that create, post, and wait on a semaphore in
34505 the file **/tmp/semaphore**. Once the file is created, the post and wait programs increment and
34506 decrement the counting semaphore (waiting and waking as required) even though they did not
34507 initialize the semaphore.

```
34508       /* create.c */
34509       #include "pthread.h"
34510       #include "sem.h"

34511       int
34512       main()
34513       {
34514           semaphore_t *semap;

34515           semap = semaphore_create("/tmp/semaphore");
34516           if (semap == NULL)
34517               exit(1);
34518           semaphore_close(semap);
34519           return (0);
34520       }

34521       /* post */
34522       #include "pthread.h"
34523       #include "sem.h"

34524       int
34525       main()
34526       {
34527           semaphore_t *semap;
```

```
            semap = semaphore_open("/tmp/semaphore");
            if (semap == NULL)
                exit(1);
            semaphore_post(semap);
            semaphore_close(semap);
            return (0);
        }

        /* wait */
        #include "pthread.h"
        #include "sem.h"

        int
        main()
        {
            semaphore_t *semap;

            semap = semaphore_open("/tmp/semaphore");
            if (semap == NULL)
                exit(1);
            semaphore_wait(semap);
            semaphore_close(semap);
            return (0);
        }
```

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy( ), *pthread_create*( ), *pthread_mutex_destroy*( ), *pthread_mutexattr_destroy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_mutexattr_destroy*( ) and *pthread_mutexattr_init*( ) functions are marked as part of the Threads option.

IEEE PASC Interpretation 1003.1c #27 is applied, updating the ERRORS section.

34560 **NAME**

34561 pthread_mutexattr_getprioceiling, pthread_mutexattr_setprioceiling — get and set prioceiling
34562 attribute of mutex attributes object (**REALTIME THREADS**)

34563 **SYNOPSIS**

34564 THR TPP
```
#include <pthread.h>

int pthread_mutexattr_getprioceiling(
    const pthread_mutexattr_t *restrict attr,
    int *restrict prioceiling);
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
    int prioceiling);
```
34570

34571 **DESCRIPTION**

34572 The *pthread_mutexattr_getprioceiling*( ) and *pthread_mutexattr_setprioceiling*( ) functions,
34573 respectively, shall get and set the priority ceiling attribute of a mutex attributes object pointed to
34574 by *attr* which was previously created by the function *pthread_mutexattr_init*( ).

34575 The *prioceiling* attribute contains the priority ceiling of initialized mutexes. The values of
34576 *prioceiling* are within the maximum range of priorities defined by SCHED_FIFO.

34577 The *prioceiling* attribute defines the priority ceiling of initialized mutexes, which is the minimum
34578 priority level at which the critical section guarded by the mutex is executed. In order to avoid
34579 priority inversion, the priority ceiling of the mutex shall be set to a priority higher than or equal |
34580 to the highest priority of all the threads that may lock that mutex. The values of *prioceiling* are |
34581 within the maximum range of priorities defined under the SCHED_FIFO scheduling policy.

34582 **RETURN VALUE**

34583 Upon successful completion, the *pthread_mutexattr_getprioceiling*( ) and
34584 *pthread_mutexattr_setprioceiling*( ) functions shall return zero; otherwise, an error number shall be
34585 returned to indicate the error.

34586 **ERRORS**

34587 The *pthread_mutexattr_getprioceiling*( ) and *pthread_mutexattr_setprioceiling*( ) functions may fail if:

34588 [EINVAL] The value specified by *attr* or *prioceiling* is invalid.

34589 [EPERM] The caller does not have the privilege to perform the operation.

34590 These functions shall not return an error code of [EINTR].

34591 **EXAMPLES**

34592 None.

34593 **APPLICATION USAGE**

34594 None.

34595 **RATIONALE**

34596 None.

34597 **FUTURE DIRECTIONS**

34598 None.

34599 **SEE ALSO**

34600 *pthread_cond_destroy*( ), *pthread_create*( ), *pthread_mutex_destroy*( ), the Base Definitions volume of
34601 IEEE Std 1003.1-200x, **<pthread.h>**

34602 **CHANGE HISTORY**

34603 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34604 Marked as part of the Realtime Threads Feature Group.

34605 **Issue 6**

34606 The *pthread_mutexattr_getprioceiling*( ) and *pthread_mutexattr_setprioceiling*( ) functions are
34607 marked as part of the Threads and Thread Priority Protection options.

34608 The [ENOSYS] error condition has been removed as stubs need not be provided if an
34609 implementation does not support the Thread Priority Protection option.

34610 The [ENOTSUP] error condition has been removed since these functions do not have a *protocol*
34611 argument.

34612 The **restrict** keyword is added to the *pthread_mutexattr_getprioceiling*( ) prototype for alignment
34613 with the ISO / IEC 9899: 1999 standard.

NAME

pthread_mutexattr_getprotocol, pthread_mutexattr_setprotocol — get and set protocol attribute of mutex attributes object (**REALTIME THREADS**)

SYNOPSIS

THR
```
#include <pthread.h>
```

TPP|TPI
```
int pthread_mutexattr_getprotocol(
    const pthread_mutexattr_t *restrict attr,
    int *restrict protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
    int protocol);
```

DESCRIPTION

The *pthread_mutexattr_getprotocol*( ) and *pthread_mutexattr_setprotocol*( ) functions, respectively, shall get and set the protocol attribute of a mutex attributes object pointed to by *attr* which was previously created by the function *pthread_mutexattr_init*( ).

The *protocol* attribute defines the protocol to be followed in utilizing mutexes. The value of *protocol* may be one of:

PTHREAD_PRIO_NONE
TPI PTHREAD_PRIO_INHERIT
TPP PTHREAD_PRIO_PROTECT

which are defined in the **<pthread.h>** header.

When a thread owns a mutex with the PTHREAD_PRIO_NONE *protocol* attribute, its priority and scheduling shall not be affected by its mutex ownership.

TPI When a thread is blocking higher priority threads because of owning one or more mutexes with the PTHREAD_PRIO_INHERIT protocol attribute, it shall execute at the higher of its priority or the priority of the highest priority thread waiting on any of the mutexes owned by this thread and initialized with this protocol.

TPP When a thread owns one or more mutexes initialized with the PTHREAD_PRIO_PROTECT protocol, it shall execute at the higher of its priority or the highest of the priority ceilings of all the mutexes owned by this thread and initialized with this attribute, regardless of whether other threads are blocked on any of these mutexes or not.

While a thread is holding a mutex which has been initialized with the PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT protocol attributes, it shall not be subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed, such as by a call to *sched_setparam*( ). Likewise, when a thread unlocks a mutex that has been initialized with the PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT protocol attributes, it shall not be subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed.

If a thread simultaneously owns several mutexes initialized with different protocols, it shall execute at the highest of the priorities that it would have obtained by each of these protocols.

TPI When a thread makes a call to *pthread_mutex_lock*( ), the mutex was initialized with the protocol attribute having the value PTHREAD_PRIO_INHERIT, when the calling thread is blocked because the mutex is owned by another thread, that owner thread shall inherit the priority level of the calling thread as long as it continues to own the mutex. The implementation shall update its execution priority to the maximum of its assigned priority and all its inherited priorities.

Furthermore, if this owner thread itself becomes blocked on another mutex, the same priority inheritance effect shall be propagated to this other owner thread, in a recursive manner.

RETURN VALUE

Upon successful completion, the *pthread_mutexattr_getprotocol*( ) and *pthread_mutexattr_setprotocol*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutexattr_setprotocol*( ) function shall fail if:

[ENOTSUP] The value specified by *protocol* is an unsupported value.

The *pthread_mutexattr_getprotocol*( ) and *pthread_mutexattr_setprotocol*( ) functions may fail if:

[EINVAL] The value specified by *attr* or *protocol* is invalid.

[EPERM] The caller does not have the privilege to perform the operation.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy( ), *pthread_create*( ), *pthread_mutex_destroy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_mutexattr_getprotocol*( ) and *pthread_mutexattr_setprotocol*( ) functions are marked as part of the Threads option and either the Thread Priority Protection or Thread Priority Inheritance options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Priority Protection or Thread Priority Inheritance options.

The **restrict** keyword is added to the *pthread_mutexattr_getprotocol*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

34696 **NAME**

34697 pthread_mutexattr_getpshared, pthread_mutexattr_setpshared — get and set process-shared
34698 attribute

34699 **SYNOPSIS**

34700 THR TSH
```
#include <pthread.h>

int pthread_mutexattr_getpshared(
    const pthread_mutexattr_t *restrict attr,
    int *restrict pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
    int pshared);
```
34706

34707 **DESCRIPTION**

34708 The *pthread_mutexattr_getpshared*( ) function shall obtain the value of the *process-shared* attribute
34709 from the attributes object referenced by *attr*. The *pthread_mutexattr_setpshared*( ) function shall |
34710 set the *process-shared* attribute in an initialized attributes object referenced by *attr*. |

34711 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a mutex to be
34712 operated upon by any thread that has access to the memory where the mutex is allocated, even if
34713 the mutex is allocated in memory that is shared by multiple processes. If the *process-shared*
34714 attribute is PTHREAD_PROCESS_PRIVATE, the mutex shall only be operated upon by threads
34715 created within the same process as the thread that initialized the mutex; if threads of differing
34716 processes attempt to operate on such a mutex, the behavior is undefined. The default value of
34717 the attribute shall be PTHREAD_PROCESS_PRIVATE.

34718 **RETURN VALUE**

34719 Upon successful completion, *pthread_mutexattr_setpshared*( ) shall return zero; otherwise, an error
34720 number shall be returned to indicate the error.

34721 Upon successful completion, *pthread_mutexattr_getpshared*( ) shall return zero and stores the
34722 value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter.
34723 Otherwise, an error number shall be returned to indicate the error.

34724 **ERRORS**

34725 The *pthread_mutexattr_getpshared*( ) and *pthread_mutexattr_setpshared*( ) functions may fail if:

34726 [EINVAL] The value specified by *attr* is invalid.

34727 The *pthread_mutexattr_setpshared*( ) function may fail if:

34728 [EINVAL] The new value specified for the attribute is outside the range of legal values
34729 for that attribute.

34730 These functions shall not return an error code of [EINTR].

34731 **EXAMPLES**

34732 None.

34733 **APPLICATION USAGE**

34734 None.

34735 **RATIONALE**

34736 None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy( ), *pthread_create*( ), *pthread_mutex_destroy*( ), *pthread_mutexattr_destroy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_mutexattr_getpshared*( ) and *pthread_mutexattr_setpshared*( ) functions are marked as part of the Threads and Thread Process-Shared Synchronization options.

The **restrict** keyword is added to the *pthread_mutexattr_getpshared*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

34749 **NAME**

34750 pthread_mutexattr_gettype, pthread_mutexattr_settype — get and set a mutex type attribute

34751 **SYNOPSIS**

34752 XSI
```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
    int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

34756

34757 **DESCRIPTION**

34758 The *pthread_mutexattr_gettype*( ) and *pthread_mutexattr_settype*( ) functions, respectively, shall get
34759 and set the mutex *type* attribute. This attribute is set in the *type* parameter to these functions. The
34760 default value of the *type* attribute is PTHREAD_MUTEX_DEFAULT.

34761 The type of mutex is contained in the *type* attribute of the mutex attributes. Valid mutex types
34762 include:

34763 PTHREAD_MUTEX_NORMAL
34764 This type of mutex does not detect deadlock. A thread attempting to relock this mutex
34765 without first unlocking it shall deadlock. Attempting to unlock a mutex locked by a
34766 different thread results in undefined behavior. Attempting to unlock an unlocked mutex
34767 results in undefined behavior.

34768 PTHREAD_MUTEX_ERRORCHECK
34769 This type of mutex provides error checking. A thread attempting to relock this mutex
34770 without first unlocking it shall return with an error. A thread attempting to unlock a mutex
34771 which another thread has locked shall return with an error. A thread attempting to unlock
34772 an unlocked mutex shall return with an error.

34773 PTHREAD_MUTEX_RECURSIVE
34774 A thread attempting to relock this mutex without first unlocking it shall succeed in locking
34775 the mutex. The relocking deadlock which can occur with mutexes of type
34776 PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this |
34777 mutex shall require the same number of unlocks to release the mutex before another thread |
34778 can acquire the mutex. A thread attempting to unlock a mutex which another thread has |
34779 locked shall return with an error. A thread attempting to unlock an unlocked mutex shall |
34780 return with an error. |

34781 PTHREAD_MUTEX_DEFAULT
34782 Attempting to recursively lock a mutex of this type results in undefined behavior.
34783 Attempting to unlock a mutex of this type which was not locked by the calling thread
34784 results in undefined behavior. Attempting to unlock a mutex of this type which is not
34785 locked results in undefined behavior. An implementation may map this mutex to one of the |
34786 other mutex types.

34787 **RETURN VALUE**

34788 Upon successful completion, the *pthread_mutexattr_gettype*( ) function shall return zero and store
34789 the value of the *type* attribute of *attr* into the object referenced by the *type* parameter. Otherwise,
34790 an error shall be returned to indicate the error.

34791 If successful, the *pthread_mutexattr_settype*( ) function shall return zero; otherwise, an error
34792 number shall be returned to indicate the error.

34793 **ERRORS**

34794 The *pthread_mutexattr_settype*( ) function shall fail if:

34795 [EINVAL] The value *type* is invalid.

34796 The *pthread_mutexattr_gettype*( ) and *pthread_mutexattr_settype*( ) functions may fail if:

34797 [EINVAL] The value specified by *attr* is invalid.

34798 These functions shall not return an error code of [EINTR].

34799 **EXAMPLES**

34800 None.

34801 **APPLICATION USAGE**

34802 It is advised that an application should not use a PTHREAD_MUTEX_RECURSIVE mutex with
34803 condition variables because the implicit unlock performed for a *pthread_cond_timedwait*( ) or
34804 *pthread_cond_wait*( ) may not actually release the mutex (if it had been locked multiple times). If
34805 this happens, no other thread can satisfy the condition of the predicate.

34806 **RATIONALE**

34807 None.

34808 **FUTURE DIRECTIONS**

34809 None.

34810 **SEE ALSO**

34811 *pthread_cond_timedwait*( ), *pthread_cond_wait*( ), the Base Definitions volume of
34812 IEEE Std 1003.1-200x, **<pthread.h>**

34813 **CHANGE HISTORY**

34814 First released in Issue 5.

34815 **Issue 6**

34816 The Open Group Corrigendum U033/3 is applied. The SYNOPSIS for
34817 *pthread_mutexattr_gettype*( ) is updated so that the first argument is of type **const**
34818 **pthread_mutexattr_t *.**

34819 The **restrict** keyword is added to the *pthread_mutexattr_gettype*( ) prototype for alignment with
34820 the ISO/IEC 9899:1999 standard.

34821 **NAME**

34822 pthread_mutexattr_init — initialize mutex attributes object

34823 **SYNOPSIS**

34824 THR
```
#include <pthread.h>
```

34825
```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

34826

34827 **DESCRIPTION**

34828 Refer to *pthread_mutexattr_destroy*( ).

34829 **NAME**

34830 pthread_mutexattr_setprioceiling — set prioceiling attribute of mutex attributes object
34831 (**REALTIME THREADS**)

34832 **SYNOPSIS**

34833 THR TPP
```
#include <pthread.h>

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
    int prioceiling);
```
34836

34837 **DESCRIPTION**

34838 Refer to *pthread_mutexattr_getprioceiling*( ).

34839 **NAME**

34840 pthread_mutexattr_setprotocol — set protocol attribute of mutex attributes object (**REALTIME**
34841 **THREADS**)

34842 **SYNOPSIS**

34843 THR
```
#include <pthread.h>
```

34844 TPP|TPI
```
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
    int protocol);
```

34846

34847 **DESCRIPTION**

34848 Refer to *pthread_mutexattr_setprotocol*( ).

34849 **NAME**

34850 pthread_mutexattr_setpshared — set process-shared attribute

34851 **SYNOPSIS**

34852 THR TSH
```
#include <pthread.h>

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
    int pshared);
```
34853
34854
34855

34856 **DESCRIPTION**

34857 Refer to *pthread_mutexattr_getpshared*( ).

34858 **NAME**

34859 pthread_mutexattr_settype — set a mutex type attribute

34860 **SYNOPSIS**

34861 XSI
```
#include <pthread.h>
```

34862
```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

34863

34864 **DESCRIPTION**

34865 Refer to *pthread_mutexattr_gettype*( ).

34866 **NAME**

34867 pthread_once — dynamic package initialization

34868 **SYNOPSIS**

34869 THR
```
#include <pthread.h>

int pthread_once(pthread_once_t *once_control,
    void (*init_routine)(void));
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```
34873

34874 **DESCRIPTION**

34875 The first call to *pthread_once*( ) by any thread in a process, with a given *once_control*, shall call the
34876 *init_routine* with no arguments. Subsequent calls of *pthread_once*( ) with the same *once_control*
34877 shall not call the *init_routine*. On return from *pthread_once*( ), *init_routine* shall have completed. |
34878 The *once_control* parameter shall determine whether the associated initialization routine has |
34879 been called.

34880 The *pthread_once*( ) function is not a cancelation point. However, if *init_routine* is a cancelation
34881 point and is canceled, the effect on *once_control* shall be as if *pthread_once*( ) was never called.

34882 The constant PTHREAD_ONCE_INIT is defined in the **<pthread.h>** header. |

34883 The behavior of *pthread_once*( ) is undefined if *once_control* has automatic storage duration or is
34884 not initialized by PTHREAD_ONCE_INIT.

34885 **RETURN VALUE**

34886 Upon successful completion, *pthread_once*( ) shall return zero; otherwise, an error number shall
34887 be returned to indicate the error.

34888 **ERRORS**

34889 The *pthread_once*( ) function may fail if:

34890 [EINVAL] If either *once_control* or *init_routine* is invalid.

34891 The *pthread_once*( ) function shall not return an error code of [EINTR].

34892 **EXAMPLES**

34893 None.

34894 **APPLICATION USAGE**

34895 None.

34896 **RATIONALE**

34897 Some C libraries are designed for dynamic initialization. That is, the global initialization for the
34898 library is performed when the first procedure in the library is called. In a single-threaded
34899 program, this is normally implemented using a static variable whose value is checked on entry
34900 to a routine, as follows:

```
static int random_is_initialized = 0;
extern int initialize_random();

int random_function()
{
    if (random_is_initialized == 0) {
        initialize_random();
        random_is_initialized = 1;
    }
    ... /* Operations performed after initialization. */
}
```

To keep the same structure in a multi-threaded program, a new primitive is needed. Otherwise, library initialization has to be accomplished by an explicit call to a library-exported initialization function prior to any use of the library.

For dynamic library initialization in a multi-threaded process, a simple initialization flag is not sufficient; the flag needs to be protected against modification by multiple threads simultaneously calling into the library. Protecting the flag requires the use of a mutex; however, mutexes have to be initialized before they are used. Ensuring that the mutex is only initialized once requires a recursive solution to this problem.

The use of *pthread_once*() not only supplies an implementation-guaranteed means of dynamic initialization, it provides an aid to the reliable construction of multi-threaded and realtime systems. The preceding example then becomes:

```
#include <pthread.h>
static pthread_once_t random_is_initialized = PTHREAD_ONCE_INIT;
extern int initialize_random();

int random_function()
{
    (void) pthread_once(&random_is_initialized, initialize_random);
    ... /* Operations performed after initialization. */
}
```

Note that a **pthread_once_t** cannot be an array because some compilers do not accept the construct **&<array_name>**.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_once*() function is marked as part of the Threads option.

The [EINVAL] error is added as a may fail case for if either argument is invalid.

34941 **NAME**

34942 pthread_rwlock_destroy, pthread_rwlock_init — destroy and initialize a read-write lock object

34943 **SYNOPSIS**

34944 THR
```
#include <pthread.h>

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
    const pthread_rwlockattr_t *restrict attr);
```
34948

34949 **DESCRIPTION**

34950 The *pthread_rwlock_destroy*( ) function shall destroy the read-write lock object referenced by
34951 *rwlock* and release any resources used by the lock. The effect of subsequent use of the lock is |
34952 undefined until the lock is reinitialized by another call to *pthread_rwlock_init*( ). An |
34953 implementation may cause *pthread_rwlock_destroy*( ) to set the object referenced by *rwlock* to an
34954 invalid value. Results are undefined if *pthread_rwlock_destroy*( ) is called when any thread holds
34955 *rwlock*. Attempting to destroy an uninitialized read-write lock results in undefined behavior.

34956 The *pthread_rwlock_init*( ) function shall allocate any resources required to use the read-write
34957 lock referenced by *rwlock* and initializes the lock to an unlocked state with attributes referenced
34958 by *attr*. If *attr* is NULL, the default read-write lock attributes shall be used; the effect is the same |
34959 as passing the address of a default read-write lock attributes object. Once initialized, the lock can |
34960 be used any number of times without being reinitialized. Results are undefined if |
34961 *pthread_rwlock_init*( ) is called specifying an already initialized read-write lock. Results are
34962 undefined if a read-write lock is used without first being initialized.

34963 If the *pthread_rwlock_init*( ) function fails, *rwlock* shall not be initialized and the contents of *rwlock* |
34964 are undefined.

34965 Only the object referenced by *rwlock* may be used for performing synchronization. The result of
34966 referring to copies of that object in calls to *pthread_rwlock_destroy*( ), *pthread_rwlock_rdlock*( ),
34967 *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ),
34968 *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( ), or *pthread_rwlock_wrlock*( ) is undefined.

34969 **RETURN VALUE**

34970 If successful, the *pthread_rwlock_destroy*( ) and *pthread_rwlock_init*( ) functions shall return zero;
34971 otherwise, an error number shall be returned to indicate the error.

34972 The [EBUSY] and [EINVAL] error checks, if implemented, act as if they were performed
34973 immediately at the beginning of processing for the function and caused an error return prior to
34974 modifying the state of the read-write lock specified by *rwlock*.

34975 **ERRORS**

34976 The *pthread_rwlock_destroy*( ) function may fail if:

34977 [EBUSY] The implementation has detected an attempt to destroy the object referenced
34978 by *rwlock* while it is locked.

34979 [EINVAL] The value specified by *rwlock* is invalid.

34980 The *pthread_rwlock_init*( ) function shall fail if:

34981 [EAGAIN] The system lacked the necessary resources (other than memory) to initialize
34982 another read-write lock.

34983 [ENOMEM] Insufficient memory exists to initialize the read-write lock. |

34984 [EPERM] The caller does not have the privilege to perform the operation. |

The *pthread_rwlock_init*( ) function may fail if:

[EBUSY] The implementation has detected an attempt to reinitialize the object referenced by *rwlock*, a previously initialized but not yet destroyed read-write lock. |

[EINVAL] The value specified by *attr* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_rdlock( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ), *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( ), *pthread_rwlock_wrlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension). The initializer macro is also deleted from the SYNOPSIS.
- The DESCRIPTION is updated as follows:
  - — It explicitly notes allocation of resources upon initialization of a read-write lock object.
  - — A paragraph is added specifying that copies of read-write lock objects may not be used.
- An [EINVAL] error is added to the ERRORS section for *pthread_rwlock_init*( ), indicating that the *rwlock* value is invalid.
- The SEE ALSO section is updated.

The **restrict** keyword is added to the *pthread_rwlock_init*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

35019 **NAME**

35020 pthread_rwlock_init — initialize a read-write lock object

35021 **SYNOPSIS**

35022 THR
```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
    const pthread_rwlockattr_t *restrict attr);
```
35023

35024

35025

35026 **DESCRIPTION**

35027 Refer to *pthread_rwlock_destroy*( ).

35028 **NAME**

35029 pthread_rwlock_rdlock, pthread_rwlock_tryrdlock — lock a read-write lock object for reading

35030 **SYNOPSIS**

35031 THR
```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```
35034

35035 **DESCRIPTION**

35036 The *pthread_rwlock_rdlock*( ) function shall apply a read lock to the read-write lock referenced by
35037 *rwlock*. The calling thread acquires the read lock if a writer does not hold the lock and there are
35038 no writers blocked on the lock.

35039 TPS If the Thread Execution Scheduling option is supported, and the threads involved in the lock are
35040 executing with the scheduling policies SCHED_FIFO or SCHED_RR, the calling thread shall not
35041 acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on
35042 the lock; otherwise, the calling thread shall acquire the lock.

35043 TPS TSP If the Threads Execution Scheduling option is supported, and the threads involved in the lock
35044 are executing with the SCHED_SPORADIC scheduling policy, the calling thread shall not
35045 acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on
35046 the lock; otherwise, the calling thread shall acquire the lock.

35047 If the Thread Execution Scheduling option is not supported, it is implementation-defined
35048 whether the calling thread acquires the lock when a writer does not hold the lock and there are
35049 writers blocked on the lock. If a writer holds the lock, the calling thread shall not acquire the
35050 read lock. If the read lock is not acquired, the calling thread shall block until it can acquire the |
35051 lock. The calling thread may deadlock if at the time the call is made it holds a write lock. |

35052 A thread may hold multiple concurrent read locks on *rwlock* (that is, successfully call the
35053 *pthread_rwlock_rdlock*( ) function *n* times). If so, the application shall ensure that the thread
35054 performs matching unlocks (that is, it calls the *pthread_rwlock_unlock*( ) function *n* times).

35055 The maximum number of simultaneous read locks that an implementation guarantees can be
35056 applied to a read-write lock shall be implementation-defined. The *pthread_rwlock_rdlock*( )
35057 function may fail if this maximum would be exceeded.

35058 The *pthread_rwlock_tryrdlock*( ) function shall apply a read lock as in the *pthread_rwlock_rdlock*( )
35059 function, with the exception that the function shall fail if the equivalent *pthread_rwlock_rdlock*( )
35060 call would have blocked the calling thread. In no case shall the *pthread_rwlock_tryrdlock*( )
35061 function ever block; it always either acquires the lock or fails and returns immediately.

35062 Results are undefined if any of these functions are called with an uninitialized read-write lock.

35063 If a signal is delivered to a thread waiting for a read-write lock for reading, upon return from the
35064 signal handler the thread resumes waiting for the read-write lock for reading as if it was not
35065 interrupted.

35066 **RETURN VALUE**

35067 If successful, the *pthread_rwlock_rdlock*( ) function shall return zero; otherwise, an error number
35068 shall be returned to indicate the error.

35069 The *pthread_rwlock_tryrdlock*( ) function shall return zero if the lock for reading on the read-write
35070 lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to
35071 indicate the error.

35072 **ERRORS**

35073 The *pthread_rwlock_tryrdlock*( ) function shall fail if:

35074 [EBUSY] The read-write lock could not be acquired for reading because a writer holds
35075 the lock or a writer with the appropriate priority was blocked on it.

35076 The *pthread_rwlock_rdlock*( ) and *pthread_rwlock_tryrdlock*( ) functions may fail if:

35077 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
35078 object.

35079 [EAGAIN] The read lock could not be acquired because the maximum number of read
35080 locks for *rwlock* has been exceeded.

35081 The *pthread_rwlock_rdlock*( ) function may fail if:

35082 [EDEADLK] The current thread already owns the read-write lock for writing.

35083 These functions shall not return an error code of [EINTR].

35084 **EXAMPLES**

35085 None.

35086 **APPLICATION USAGE**

35087 Applications using these functions may be subject to priority inversion, as discussed in the Base
35088 Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

35089 **RATIONALE**

35090 None.

35091 **FUTURE DIRECTIONS**

35092 None.

35093 **SEE ALSO**

35094 *pthread_rwlock_destroy*( ), *pthread_rwlock_init*( ), *pthread_rwlock_timedrdlock*( ),
35095 *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( ),
35096 *pthread_rwlock_wrlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

35097 **CHANGE HISTORY**

35098 First released in Issue 5.

35099 **Issue 6**

35100 The following changes are made for alignment with IEEE Std 1003.1j-2000:

35101 • The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is
35102 now part of the Threads option (previously it was part of the Read-Write Locks option in
35103 IEEE Std 1003.1j-2000 and also part of the XSI extension).

35104 • The DESCRIPTION is updated as follows:

35105 — Conditions under which writers have precedence over readers are specified.

35106 — Failure of *pthread_rwlock_tryrdlock*( ) is clarified.

35107 — A paragraph on the maximum number of read locks is added.

35108 • In the ERRORS sections, [EBUSY] is modified to take into account write priority, and
35109 [EDEADLK] is deleted as a *pthread_rwlock_tryrdlock*( ) error.

35110 • The SEE ALSO section is updated.

35111 **NAME**

35112 pthread_rwlock_timedrdlock — lock a read-write lock for reading

35113 **SYNOPSIS**

35114 THR TMO
```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
    const struct timespec *restrict abs_timeout);
```
35118

35119 **DESCRIPTION**

35120 The *pthread_rwlock_timedrdlock*( ) function shall apply a read lock to the read-write lock
35121 referenced by *rwlock* as in the *pthread_rwlock_rdlock*( ) function. However, if the lock cannot be
35122 acquired without waiting for other threads to unlock the lock, this wait shall be terminated
35123 when the specified timeout expires. The timeout shall expire when the absolute time specified |
35124 by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the |
35125 value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout*
35126 has already been passed at the time of the call.

35127 TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock. If |
35128 the Timers option is not supported, the timeout shall be based on the system clock as returned |
35129 by the *time*( ) function. The resolution of the timeout shall be the resolution of the clock on which |
35130 it is based. The **timespec** data type is defined in the **<time.h>** header. Under no circumstances |
35131 shall the function fail with a timeout if the lock can be acquired immediately. The validity of the
35132 *abs_timeout* parameter need not be checked if the lock can be immediately acquired.

35133 If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-
35134 write lock via a call to *pthread_rwlock_timedrdlock*( ), upon return from the signal handler the
35135 thread shall resume waiting for the lock as if it was not interrupted.

35136 The calling thread may deadlock if at the time the call is made it holds a write lock on *rwlock*.
35137 The results are undefined if this function is called with an uninitialized read-write lock.

35138 **RETURN VALUE**

35139 The *pthread_rwlock_timedrdlock*( ) function shall return zero if the lock for reading on the read-
35140 write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned
35141 to indicate the error.

35142 **ERRORS**

35143 The *pthread_rwlock_timedrdlock*( ) function shall fail if:

35144 [ETIMEDOUT] The lock could not be acquired before the specified timeout expired.

35145 The *pthread_rwlock_timedrdlock*( ) function may fail if:

35146 [EAGAIN] The read lock could not be acquired because the maximum number of read
35147 locks for lock would be exceeded.

35148 [EDEADLK] The calling thread already holds a write lock on *rwlock*.

35149 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
35150 object, or the *abs_timeout* nanosecond value is less than zero or greater than or
35151 equal to 1 000 million.

35152 This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications using this function may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

The *pthread_rwlock_timedrdlock*( ) function is part of the Threads and Timeouts options and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy( ), *pthread_rwlock_init*( ), *pthread_rwlock_rdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ), *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( ), *pthread_rwlock_wrlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

35171 **NAME**

35172 pthread_rwlock_timedwrlock — lock a read-write lock for writing

35173 **SYNOPSIS**

35174 THR TMO
```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
    const struct timespec *restrict abs_timeout);
```
35178

35179 **DESCRIPTION**

35180 The *pthread_rwlock_timedwrlock*( ) function shall apply a write lock to the read-write lock
35181 referenced by *rwlock* as in the *pthread_rwlock_wrlock*( ) function. However, if the lock cannot be
35182 acquired without waiting for other threads to unlock the lock, this wait shall be terminated
35183 when the specified timeout expires. The timeout shall expire when the absolute time specified |
35184 by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the |
35185 value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout*
35186 has already been passed at the time of the call.

35187 TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock. If |
35188 the Timers option is not supported, the timeout shall be based on the system clock as returned |
35189 by the *time*( ) function. The resolution of the timeout shall be the resolution of the clock on which |
35190 it is based. The **timespec** data type is defined in the **<time.h>** header. Under no circumstances |
35191 shall the function fail with a timeout if the lock can be acquired immediately. The validity of the
35192 *abs_timeout* parameter need not be checked if the lock can be immediately acquired.

35193 If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-
35194 write lock via a call to *pthread_rwlock_timedwrlock*( ), upon return from the signal handler the
35195 thread shall resume waiting for the lock as if it was not interrupted.

35196 The calling thread may deadlock if at the time the call is made it holds the read-write lock. The
35197 results are undefined if this function is called with an uninitialized read-write lock.

35198 **RETURN VALUE**

35199 The *pthread_rwlock_timedwrlock*( ) function shall return zero if the lock for writing on the read-
35200 write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned
35201 to indicate the error.

35202 **ERRORS**

35203 The *pthread_rwlock_timedwrlock*( ) function shall fail if:

35204 [ETIMEDOUT] The lock could not be acquired before the specified timeout expired.

35205 The *pthread_rwlock_timedwrlock*( ) function may fail if:

35206 [EDEADLK] The calling thread already holds the *rwlock*.

35207 [EINVAL] The value specified by rwlock does not refer to an initialized read-write lock
35208 object, or the *abs_timeout* nanosecond value is less than zero or greater than or
35209 equal to 1 000 million.

35210 This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications using this function may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

The *pthread_rwlock_timedwrlock*( ) function is part of the Threads and Timeouts options and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy( ), *pthread_rwlock_init*( ), *pthread_rwlock_rdlock*( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_tryrdlock*( ), *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( ), *pthread_rwlock_wrlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

35229 **NAME**

35230 pthread_rwlock_tryrdlock — lock a read-write lock object for reading

35231 **SYNOPSIS**

35232 THR
```
#include <pthread.h>
```

35233
```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

35234

35235 **DESCRIPTION**

35236 Refer to *pthread_rwlock_rdlock*( ).

35237 **NAME**

35238 pthread_rwlock_trywrlock, pthread_rwlock_wrlock — lock a read-write lock object for writing

35239 **SYNOPSIS**

35240 THR
```
#include <pthread.h>

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```
35243

35244 **DESCRIPTION**

35245 The *pthread_rwlock_trywrlock*( ) function shall apply a write lock like the *pthread_rwlock_wrlock*( )
35246 function, with the exception that the function shall fail if any thread currently holds *rwlock* (for
35247 reading or writing).

35248 The *pthread_rwlock_wrlock*( ) function shall apply a write lock to the read-write lock referenced
35249 by *rwlock*. The calling thread acquires the write lock if no other thread (reader or writer) holds
35250 the read-write lock *rwlock*. Otherwise, the thread shall block until it can acquire the lock. The |
35251 calling thread may deadlock if at the time the call is made it holds the read-write lock (whether a |
35252 read or write lock). |

35253 Implementations may favor writers over readers to avoid writer starvation. |

35254 Results are undefined if any of these functions are called with an uninitialized read-write lock.

35255 If a signal is delivered to a thread waiting for a read-write lock for writing, upon return from the
35256 signal handler the thread resumes waiting for the read-write lock for writing as if it was not
35257 interrupted.

35258 **RETURN VALUE**

35259 The *pthread_rwlock_trywrlock*( ) function shall return zero if the lock for writing on the read-write
35260 lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to
35261 indicate the error.

35262 If successful, the *pthread_rwlock_wrlock*( ) function shall return zero; otherwise, an error number
35263 shall be returned to indicate the error.

35264 **ERRORS**

35265 The *pthread_rwlock_trywrlock*( ) function shall fail if:

35266 [EBUSY] The read-write lock could not be acquired for writing because it was already
35267 locked for reading or writing.

35268 The *pthread_rwlock_trywrlock*( ) and *pthread_rwlock_wrlock*( ) functions may fail if:

35269 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
35270 object.

35271 The *pthread_rwlock_wrlock*( ) function may fail if:

35272 [EDEADLK] The current thread already owns the read-write lock for writing or reading.

35273 These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy( ), *pthread_rwlock_init*( ), *pthread_rwlock_rdlock*( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ), *pthread_rwlock_unlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).
- The [EDEADLK] error is deleted as a *pthread_rwlock_trywrlock*( ) error.
- The SEE ALSO section is updated.

35296 **NAME**

35297 pthread_rwlock_unlock — unlock a read-write lock object

35298 **SYNOPSIS**

35299 THR
```
#include <pthread.h>
```

35300
```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

35301

35302 **DESCRIPTION**

35303 The *pthread_rwlock_unlock*( ) function shall release a lock held on the read-write lock object |
35304 referenced by *rwlock*. Results are undefined if the read-write lock *rwlock* is not held by the
35305 calling thread.

35306 If this function is called to release a read lock from the read-write lock object and there are other
35307 read locks currently held on this read-write lock object, the read-write lock object remains in the
35308 read locked state. If this function releases the last read lock for this read-write lock object, the
35309 read-write lock object shall be put in the unlocked state with no owners.

35310 If this function is called to release a write lock for this read-write lock object, the read-write lock
35311 object shall be put in the unlocked state.

35312 If there are threads blocked on the lock when it becomes available, the scheduling policy shall |
35313 TPS determine which thread(s) shall acquire the lock. If the Thread Execution Scheduling option is |
35314 supported, when threads executing with the scheduling policies SCHED_FIFO, SCHED_RR, or |
35315 SCHED_SPORADIC are waiting on the lock, they shall acquire the lock in priority order when |
35316 the lock becomes available. For equal priority threads, write locks shall take precedence over |
35317 read locks. If the Thread Execution Scheduling option is not supported, it is implementation- |
35318 defined whether write locks take precedence over read locks.

35319 Results are undefined if any of these functions are called with an uninitialized read-write lock.

35320 **RETURN VALUE**

35321 If successful, the *pthread_rwlock_unlock*( ) function shall return zero; otherwise, an error number
35322 shall be returned to indicate the error.

35323 **ERRORS**

35324 The *pthread_rwlock_unlock*( ) function may fail if:

35325 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
35326 object.

35327 [EPERM] The current thread does not hold a lock on the read-write lock.

35328 The *pthread_rwlock_unlock*( ) function shall not return an error code of [EINTR].

35329 **EXAMPLES**

35330 None.

35331 **APPLICATION USAGE**

35332 None.

35333 **RATIONALE**

35334 None.

35335 **FUTURE DIRECTIONS**

35336 None.

35337 **SEE ALSO**

35338 *pthread_rwlock_destroy*( ), *pthread_rwlock_init*( ), *pthread_rwlock_rdlock*( ),
35339 *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ),
35340 *pthread_rwlock_trywrlock*( ), *pthread_rwlock_wrlock*( ), the Base Definitions volume of
35341 IEEE Std 1003.1-200x, **<pthread.h>**

35342 **CHANGE HISTORY**

35343 First released in Issue 5.

35344 **Issue 6**

35345 The following changes are made for alignment with IEEE Std 1003.1j-2000:

35346 • The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is
35347 now part of the Threads option (previously it was part of the Read-Write Locks option in
35348 IEEE Std 1003.1j-2000 and also part of the XSI extension).

35349 • The DESCRIPTION is updated as follows:

35350 — The conditions under which writers have precedence over readers are specified.

35351 — The concept of read-write lock owner is deleted.

35352 • The SEE ALSO section is updated.

35353 **NAME**

35354 pthread_rwlock_wrlock — lock a read-write lock object for writing

35355 **SYNOPSIS**

35356 THR
```
#include <pthread.h>

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```
35357

35358

35359 **DESCRIPTION**

35360 Refer to *pthread_rwlock_trywrlock*( ).

35361 **NAME**

35362 pthread_rwlockattr_destroy, pthread_rwlockattr_init — destroy and initialize read-write lock
35363 attributes object

35364 **SYNOPSIS**

35365 THR
```
#include <pthread.h>

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```
35368

35369 **DESCRIPTION**

35370 The *pthread_rwlockattr_destroy*( ) function shall destroy a read-write lock attributes object. A |
35371 destroyed *attr* attributes object can be reinitialized using *pthread_rwlockattr_init*( ); the results of |
35372 otherwise referencing the object after it has been destroyed are undefined. An implementation |
35373 may cause *pthread_rwlockattr_destroy*( ) to set the object referenced by *attr* to an invalid value. |

35374 The *pthread_rwlockattr_init*( ) function shall initialize a read-write lock attributes object *attr* with
35375 the default value for all of the attributes defined by the implementation.

35376 Results are undefined if *pthread_rwlockattr_init*( ) is called specifying an already initialized *attr* |
35377 attributes object. |

35378 After a read-write lock attributes object has been used to initialize one or more read-write locks,
35379 any function affecting the attributes object (including destruction) shall not affect any previously |
35380 initialized read-write locks.

35381 **RETURN VALUE**

35382 If successful, the *pthread_rwlockattr_destroy*( ) and *pthread_rwlockattr_init*( ) functions shall return
35383 zero; otherwise, an error number shall be returned to indicate the error.

35384 **ERRORS**

35385 The *pthread_rwlockattr_destroy*( ) function may fail if:

35386 [EINVAL] The value specified by *attr* is invalid.

35387 The *pthread_rwlockattr_init*( ) function shall fail if:

35388 [ENOMEM] Insufficient memory exists to initialize the read-write lock attributes object.

35389 These functions shall not return an error code of [EINTR].

35390 **EXAMPLES**

35391 None.

35392 **APPLICATION USAGE**

35393 None.

35394 **RATIONALE**

35395 None.

35396 **FUTURE DIRECTIONS**

35397 None.

35398 **SEE ALSO**

35399 *pthread_rwlock_init*( ), *pthread_rwlockattr_getpshared*( ), *pthread_rwlockattr_setpshared*( ), the Base
35400 Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

35401 **CHANGE HISTORY**

35402 First released in Issue 5.

35403 **Issue 6**

35404 The following changes are made for alignment with IEEE Std 1003.1j-2000:

35405 • The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is
35406 now part of the Threads option (previously it was part of the Read-Write Locks option in
35407 IEEE Std 1003.1j-2000 and also part of the XSI extension).

35408 • The SEE ALSO section is updated.

35409 **NAME**

35410 pthread_rwlockattr_getpshared, pthread_rwlockattr_setpshared — get and set process-shared
35411 attribute of read-write lock attributes object

35412 **SYNOPSIS**

35413 THR TSH
```
#include <pthread.h>

int pthread_rwlockattr_getpshared(
    const pthread_rwlockattr_t *restrict attr,
    int *restrict pshared);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
    int pshared);
```
35419

35420 **DESCRIPTION**

35421 The *pthread_rwlockattr_getpshared*( ) function shall obtain the value of the *process-shared* attribute
35422 from the initialized attributes object referenced by *attr*. The *pthread_rwlockattr_setpshared*( )
35423 function shall set the *process-shared* attribute in an initialized attributes object referenced by *attr*.

35424 The *process-shared* attribute shall be set to PTHREAD_PROCESS_SHARED to permit a read-
35425 write lock to be operated upon by any thread that has access to the memory where the read-
35426 write lock is allocated, even if the read-write lock is allocated in memory that is shared by
35427 multiple processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the read-
35428 write lock shall only be operated upon by threads created within the same process as the thread
35429 that initialized the read-write lock; if threads of differing processes attempt to operate on such a
35430 read-write lock, the behavior is undefined. The default value of the *process-shared* attribute shall
35431 be PTHREAD_PROCESS_PRIVATE.

35432 Additional attributes, their default values, and the names of the associated functions to get and
35433 set those attribute values are implementation-defined.

35434 **RETURN VALUE**

35435 Upon successful completion, the *pthread_rwlockattr_getpshared*( ) shall return zero and store the
35436 value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter.
35437 Otherwise, an error number shall be returned to indicate the error.

35438 If successful, the *pthread_rwlockattr_setpshared*( ) function shall return zero; otherwise, an error
35439 number shall be returned to indicate the error.

35440 **ERRORS**

35441 The *pthread_rwlockattr_getpshared*( ) and *pthread_rwlockattr_setpshared*( ) functions may fail if:

35442 [EINVAL] The value specified by *attr* is invalid.

35443 The *pthread_rwlockattr_setpshared*( ) function may fail if:

35444 [EINVAL] The new value specified for the attribute is outside the range of legal values
35445 for that attribute.

35446 These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_init( ), *pthread_rwlockattr_destroy*( ), *pthread_rwlockattr_init*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- The margin code in the SYNOPSIS is changed to THR TSH to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).
- The DESCRIPTION notes that additional attributes are implementation-defined.
- The SEE ALSO section is updated.

The **restrict** keyword is added to the *pthread_rwlockattr_getpshared*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

35469 **NAME**

35470 pthread_rwlockattr_init — initialize read-write lock attributes object

35471 **SYNOPSIS**

35472 XSI
```
#include <pthread.h>
```

35473
```
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

35474

35475 **DESCRIPTION**

35476 Refer to *pthread_rwlockattr_destroy*( ).

35477 **NAME**

35478 pthread_rwlockattr_setpshared — set process-shared attribute of read-write lock attributes
35479 object

35480 **SYNOPSIS**

35481 XSI
```
#include <pthread.h>

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
    int pshared);
```
35482
35483
35484

35485 **DESCRIPTION**

35486 Refer to *pthread_rwlockattr_getpshared*( ).

35487 **NAME**

35488 pthread_self — get calling thread's ID

35489 **SYNOPSIS**

35490 THR
```
#include <pthread.h>
```

35491
```
pthread_t pthread_self(void);
```

35492

35493 **DESCRIPTION**

35494 The *pthread_self*( ) function shall return the thread ID of the calling thread.

35495 **RETURN VALUE**

35496 Refer to the DESCRIPTION.

35497 **ERRORS**

35498 No errors are defined.

35499 The *pthread_self*( ) function shall not return an error code of [EINTR].

35500 **EXAMPLES**

35501 None.

35502 **APPLICATION USAGE**

35503 None.

35504 **RATIONALE**

35505 The *pthread_self*( ) function provides a capability similar to the *getpid*( ) function for processes
35506 and the rationale is the same: the creation call does not provide the thread ID to the created
35507 thread.

35508 **FUTURE DIRECTIONS**

35509 None.

35510 **SEE ALSO**

35511 *pthread_create*( ), *pthread_equal*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
35512 **<pthread.h>**

35513 **CHANGE HISTORY**

35514 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35515 **Issue 6**

35516 The *pthread_self*( ) function is marked as part of the Threads option.

35517 **NAME**

35518 pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel — set cancelability state

35519 **SYNOPSIS**

35520 THR
```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```
35524

35525 **DESCRIPTION**

35526 The *pthread_setcancelstate*( ) function shall atomically both set the calling thread's cancelability
35527 state to the indicated *state* and return the previous cancelability state at the location referenced
35528 by *oldstate*. Legal values for *state* are PTHREAD_CANCEL_ENABLE and
35529 PTHREAD_CANCEL_DISABLE.

35530 The *pthread_setcanceltype*( ) function shall atomically both set the calling thread's cancelability
35531 type to the indicated *type* and return the previous cancelability type at the location referenced by
35532 *oldtype*. Legal values for *type* are PTHREAD_CANCEL_DEFERRED and
35533 PTHREAD_CANCEL_ASYNCHRONOUS.

35534 The cancelability state and type of any newly created threads, including the thread in which
35535 *main*( ) was first invoked, shall be PTHREAD_CANCEL_ENABLE and
35536 PTHREAD_CANCEL_DEFERRED respectively.

35537 The *pthread_testcancel*( ) function shall create a cancelation point in the calling thread. The
35538 *pthread_testcancel*( ) function shall have no effect if cancelability is disabled.

35539 **RETURN VALUE**

35540 If successful, the *pthread_setcancelstate*( ) and *pthread_setcanceltype*( ) functions shall return zero;
35541 otherwise, an error number shall be returned to indicate the error.

35542 **ERRORS**

35543 The *pthread_setcancelstate*( ) function may fail if:

35544 [EINVAL] The specified state is not PTHREAD_CANCEL_ENABLE or
35545 PTHREAD_CANCEL_DISABLE.

35546 The *pthread_setcanceltype*( ) function may fail if:

35547 [EINVAL] The specified type is not PTHREAD_CANCEL_DEFERRED or
35548 PTHREAD_CANCEL_ASYNCHRONOUS.

35549 These functions shall not return an error code of [EINTR].

35550 **EXAMPLES**

35551 None.

35552 **APPLICATION USAGE**

35553 None.

35554 **RATIONALE**

35555 The *pthread_setcancelstate*( ) and *pthread_setcanceltype*( ) functions control the points at which a |
35556 thread may be asynchronously canceled. For cancelation control to be usable in modular fashion, |
35557 some rules need to be followed.

35558 An object can be considered to be a generalization of a procedure. It is a set of procedures and
35559 global variables written as a unit and called by clients not known by the object. Objects may
35560 depend on other objects.

First, cancelability should only be disabled on entry to an object, never explicitly enabled. On exit from an object, the cancelability state should always be restored to its value on entry to the object.

This follows from a modularity argument: if the client of an object (or the client of an object that uses that object) has disabled cancelability, it is because the client does not want to be concerned about cleaning up if the thread is canceled while executing some sequence of actions. If an object is called in such a state and it enables cancelability and a cancelation request is pending for that thread, then the thread is canceled, contrary to the wish of the client that disabled.

Second, the cancelability type may be explicitly set to either *deferred* or *asynchronous* upon entry to an object. But as with the cancelability state, on exit from an object the cancelability type should always be restored to its value on entry to the object.

Finally, only functions that are cancel-safe may be called from a thread that is asynchronously cancelable.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cancel( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_setcancelstate*( ), *pthread_setcanceltype*( ), and *pthread_testcancel*( ) functions are marked as part of the Threads option.

35583 **NAME**

35584 pthread_setconcurrency — set level of concurrency

35585 **SYNOPSIS**

35586 XSI
```
#include <pthread.h>
```

35587
```
int pthread_setconcurrency(int new_level);
```

35588

35589 **DESCRIPTION**

35590 Refer to *pthread_getconcurrency*( ).

35591 **NAME**

35592 pthread_setschedparam — dynamic thread scheduling parameters access (**REALTIME**
35593 **THREADS**)

35594 **SYNOPSIS**

35595 THR TPS
```
#include <pthread.h>

int pthread_setschedparam(pthread_t thread, int policy,
    const struct sched_param *param);
```
35598

35599 **DESCRIPTION**

35600 Refer to *pthread_getschedparam*( ).

35601 **NAME**

35602 pthread_setschedprio — dynamic thread scheduling parameters access (**REALTIME**
35603 **THREADS**)

35604 **SYNOPSIS**

35605 THR TPS
```
#include <pthread.h>

int pthread_setschedprio(pthread_t thread, int prio);
```
35606
35607

35608 **DESCRIPTION**

35609 The *pthread_setschedprio*( ) function shall set the scheduling priority for the thread whose thread
35610 ID is given by *thread* to the value given by *prio*. See **Scheduling Policies** (on page 494) for a
35611 description on how this function call affects the ordering of the thread in the thread list for its
35612 new priority.

35613 If the *pthread_setschedprio*( ) function fails, the scheduling priority of the target thread shall not be
35614 changed.

35615 **RETURN VALUE**

35616 If successful, the *pthread_setschedprio*( ) function shall return zero; otherwise, an error number
35617 shall be returned to indicate the error.

35618 **ERRORS**

35619 The *pthread_setschedprio*( ) function may fail if:

35620 [EINVAL] The value of *prio* is invalid for the scheduling policy of the specified thread.

35621 [ENOTSUP] An attempt was made to set the priority to an unsupported value.

35622 [EPERM] The caller does not have the appropriate permission to set the scheduling
35623 policy of the specified thread.

35624 [EPERM] The implementation does not allow the application to modify the priority to
35625 the value specified.

35626 [ESRCH] The value specified by *thread* does not refer to an existing thread.

35627 The *pthread_setschedprio*( ) function shall not return an error code of [EINTR].

35628 **EXAMPLES**

35629 None.

35630 **APPLICATION USAGE**

35631 None.

35632 **RATIONALE**

35633 The *pthread_setschedprio*( ) function provides a way for an application to temporarily raise its
35634 priority and then lower it again, without having the undesired side effect of yielding to other
35635 threads of the same priority. This is necessary if the application is to implement its own
35636 strategies for bounding priority inversion, such as priority inheritance or priority ceilings. This
35637 capability is especially important if the implementation does not support the Thread Priority
35638 Protection or Thread Priority Inheritance options, but even if those options are supported it is
35639 needed if the application is to bound priority inheritance for other resources, such as
35640 semaphores.

35641 The standard developers considered that while it might be preferable conceptually to solve this
35642 problem by modifying the specification of *pthread_setschedparam*( ), it was too late to make such a
35643 change, as there may be implementations that would need to be changed. Therefore, this new
35644 function was introduced.

35645 **FUTURE DIRECTIONS**
35646 None.

35647 **SEE ALSO**
35648 *pthread_getschedparam*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

35649 **CHANGE HISTORY**
35650 First released in Issue 6. Included as a response to IEEE PASC Interpretation 1003.1 #96.

35651 **NAME**

35652 pthread_setspecific — thread-specific data management

35653 **SYNOPSIS**

35654 THR
```
#include <pthread.h>

int pthread_setspecific(pthread_key_t key, const void *value);
```
35655

35656

35657 **DESCRIPTION**

35658 Refer to *pthread_getspecific*( ).

35659 **NAME**

35660 pthread_sigmask, sigprocmask — examine and change blocked signals

35661 **SYNOPSIS**

35662 ```
#include <signal.h>
```

35663 THR
```
int pthread_sigmask(int how, const sigset_t *restrict set,
    sigset_t *restrict oset);
```
35665 CX
```
int sigprocmask(int how, const sigset_t *restrict set,
    sigset_t *restrict oset);
```

35667

35668 **DESCRIPTION**

35669 THR The *pthread_sigmask*( ) function shall examine or change (or both) the calling thread's signal mask, regardless of the number of threads in the process. The function shall be equivalent to *sigprocmask*( ), without the restriction that the call be made in a single-threaded process.

35672 In a single-threaded process, the *sigprocmask*( ) function shall examine or change (or both) the signal mask of the calling thread.

35674 If the argument *set* is not a null pointer, it points to a set of signals to be used to change the currently blocked set.

35676 The argument *how* indicates the way in which the set is changed, and the application shall ensure it consists of one of the following values:

35678 SIG_BLOCK The resulting set shall be the union of the current set and the signal set pointed to by *set*.

35680 SIG_SETMASK The resulting set shall be the signal set pointed to by *set*.

35681 SIG_UNBLOCK The resulting set shall be the intersection of the current set and the complement of the signal set pointed to by *set*.

35683 If the argument *oset* is not a null pointer, the previous mask shall be stored in the location pointed to by *oset*. If *set* is a null pointer, the value of the argument *how* is not significant and the process' signal mask shall be unchanged; thus the call can be used to enquire about currently blocked signals.

35687 If there are any pending unblocked signals after the call to *sigprocmask*( ), at least one of those signals shall be delivered before the call to *sigprocmask*( ) returns.

35689 It is not possible to block those signals which cannot be ignored. This shall be enforced by the system without causing an error to be indicated.

35691 If any of the SIGFPE, SIGILL, SIGSEGV, or SIGBUS signals are generated while they are blocked, the result is undefined, unless the signal was generated by the *kill*( ) function, the *sigqueue*( ) function, or the *raise*( ) function.

35694 If *sigprocmask*( ) fails, the thread's signal mask shall not be changed.

35695 The use of the *sigprocmask*( ) function is unspecified in a multi-threaded process.

35696 **RETURN VALUE**

35697 THR Upon successful completion *pthread_sigmask*( ) shall return 0; otherwise, it shall return the corresponding error number.

35699 Upon successful completion, *sigprocmask*( ) shall return 0; otherwise, –1 shall be returned, *errno* shall be set to indicate the error, and the process' signal mask shall be unchanged.
```

35701 **ERRORS**

35702 THR The *pthread_sigmask*( ) and *sigprocmask*( ) functions shall fail if:

35703 [EINVAL] The value of the *how* argument is not equal to one of the defined values.

35704 THR The *pthread_sigmask*( ) function shall not return an error code of [EINTR].

35705 **EXAMPLES**

35706 None.

35707 **APPLICATION USAGE**

35708 None.

35709 **RATIONALE**

35710 When a process' signal mask is changed in a signal-catching function that is installed by
35711 *sigaction*( ), the restoration of the signal mask on return from the signal-catching function
35712 overrides that change (see *sigaction*( )). If the signal-catching function was installed with
35713 *signal*( ), it is unspecified whether this occurs.

35714 See *kill*( ) for a discussion of the requirement on delivery of signals.

35715 **FUTURE DIRECTIONS**

35716 None.

35717 **SEE ALSO**

35718 *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *sigpending*( ),
35719 *sigqueue*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

35720 **CHANGE HISTORY**

35721 First released in Issue 3.

35722 Entry included for alignment with the POSIX.1-1988 standard.

35723 **Issue 5**

35724 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

35725 The *pthread_sigmask*( ) function is added for alignment with the POSIX Threads Extension.

35726 **Issue 6**

35727 The *pthread_sigmask*( ) function is marked as part of the Threads option.

35728 The SYNOPSIS for *sigprocmask*( ) is marked as a CX extension to note that the presence of this |
35729 function in the **<signal.h>** header is an extension to the ISO C standard. |

35730 The following changes are made for alignment with the ISO POSIX-1: 1996 standard: |

35731 • The DESCRIPTION is updated to explicitly state the functions which may generate the
35732 signal.

35733 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

35734 The **restrict** keyword is added to the *pthread_sigmask*( ) and *sigprocmask*( ) prototypes for
35735 alignment with the ISO / IEC 9899: 1999 standard.

35736 **NAME**

35737 pthread_spin_destroy, pthread_spin_init — destroy or initialize a spin lock object (**ADVANCED**
35738 **REALTIME THREADS**)

35739 **SYNOPSIS**

35740 THR SPI
```
#include <pthread.h>

int pthread_spin_destroy(pthread_spinlock_t *lock);
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```
35743

35744 **DESCRIPTION**

35745 The *pthread_spin_destroy*( ) function shall destroy the spin lock referenced by *lock* and release any
35746 resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is
35747 reinitialized by another call to *pthread_spin_init*( ). The results are undefined if
35748 *pthread_spin_destroy*( ) is called when a thread holds the lock, or if this function is called with an
35749 uninitialized thread spin lock.

35750 The *pthread_spin_init*( ) function shall allocate any resources required to use the spin lock
35751 referenced by *lock* and initialize the lock to an unlocked state.

35752 TSH If the Thread Process-Shared Synchronization option is supported and the value of *pshared* is
35753 PTHREAD_PROCESS_SHARED, the implementation shall permit the spin lock to be operated
35754 upon by any thread that has access to the memory where the spin lock is allocated, even if it is
35755 allocated in memory that is shared by multiple processes.

35756 If the Thread Process-Shared Synchronization option is supported and the value of *pshared* is
35757 PTHREAD_PROCESS_PRIVATE, or if the option is not supported, the spin lock shall only be
35758 operated upon by threads created within the same process as the thread that initialized the spin
35759 lock. If threads of differing processes attempt to operate on such a spin lock, the behavior is
35760 undefined.

35761 The results are undefined if *pthread_spin_init*( ) is called specifying an already initialized spin
35762 lock. The results are undefined if a spin lock is used without first being initialized.

35763 If the *pthread_spin_init*( ) function fails, the lock is not initialized and the contents of *lock* are
35764 undefined.

35765 Only the object referenced by *lock* may be used for performing synchronization.

35766 The result of referring to copies of that object in calls to *pthread_spin_destroy*( ),
35767 *pthread_spin_lock*( ), *pthread_spin_trylock*( ), or *pthread_spin_unlock*( ) is undefined.

35768 **RETURN VALUE**

35769 Upon successful completion, these functions shall return zero; otherwise, an error number shall
35770 be returned to indicate the error.

35771 **ERRORS**

35772 These functions may fail if:

35773 [EBUSY] The implementation has detected an attempt to initialize or destroy a spin
35774 lock while it is in use (for example, while being used in a *pthread_spin_lock*( )
35775 call) by another thread.

35776 [EINVAL] The value specified by *lock* is invalid.

35777 The *pthread_spin_init*( ) function shall fail if:

35778 [EAGAIN] The system lacks the necessary resources to initialize another spin lock.

35779 [ENOMEM] Insufficient memory exists to initialize the lock.

35780 These functions shall not return an error code of [EINTR].

35781 **EXAMPLES**

35782 None.

35783 **APPLICATION USAGE**

35784 The *pthread_spin_destroy*( ) and *pthread_spin_init*( ) functions are part of the Spin Locks option
35785 and need not be provided on all implementations.

35786 **RATIONALE**

35787 None.

35788 **FUTURE DIRECTIONS**

35789 None.

35790 **SEE ALSO**

35791 *pthread_spin_lock*( ), *pthread_spin_trylock*( ), *pthread_spin_unlock*( ), the Base Definitions volume of
35792 IEEE Std 1003.1-200x, **<<pthread.h>>**

35793 **CHANGE HISTORY**

35794 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

35795 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

35796 **NAME**

35797 pthread_spin_init — initialize a spin lock object (**ADVANCED REALTIME THREADS**)

35798 **SYNOPSIS**

35799 THR SPI
```
#include <pthread.h>

int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```
35800

35801

35802 **DESCRIPTION**

35803 Refer to *pthread_spin_destroy*( ).

35804 **NAME**

35805 pthread_spin_lock, pthread_spin_trylock — lock a spin lock object (**ADVANCED REALTIME**
35806 **THREADS**)

35807 **SYNOPSIS**

35808 THR SPI
```
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
```
35809
35810
35811

35812 **DESCRIPTION**

35813 The *pthread_spin_lock*( ) function shall lock the spin lock referenced by *lock*. The calling thread
35814 shall acquire the lock if it is not held by another thread. Otherwise, the thread shall spin (that is, |
35815 shall not return from the *pthread_spin_lock*( ) call) until the lock becomes available. The results |
35816 are undefined if the calling thread holds the lock at the time the call is made. The
35817 *pthread_spin_trylock*( ) function shall lock the spin lock referenced by *lock* if it is not held by any |
35818 thread. Otherwise, the function shall fail. |

35819 The results are undefined if any of these functions is called with an uninitialized spin lock.

35820 **RETURN VALUE**

35821 Upon successful completion, these functions shall return zero; otherwise, an error number shall
35822 be returned to indicate the error.

35823 **ERRORS**

35824 These functions may fail if:

35825 [EINVAL] The value specified by *lock* does not refer to an initialized spin lock object.

35826 The *pthread_spin_lock*( ) function may fail if:

35827 [EDEADLK] The calling thread already holds the lock.

35828 The *pthread_spin_trylock*( ) function shall fail if:

35829 [EBUSY] A thread currently holds the lock.

35830 These functions shall not return an error code of [EINTR].

35831 **EXAMPLES**

35832 None.

35833 **APPLICATION USAGE**

35834 Applications using this function may be subject to priority inversion, as discussed in the Base
35835 Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

35836 The *pthread_spin_lock*( ) and *pthread_spin_trylock*( ) functions are part of the Spin Locks option
35837 and need not be provided on all implementations.

35838 **RATIONALE**

35839 None.

35840 **FUTURE DIRECTIONS**

35841 None.

35842 **SEE ALSO**

35843 *pthread_spin_init*( ), *pthread_spin_destroy*( ), *pthread_spin_unlock*( ), the Base Definitions volume of
35844 IEEE Std 1003.1-200x, **<pthread.h>**

35845 **CHANGE HISTORY**

35846 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

35847 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

35848 **NAME**

35849 pthread_spin_trylock — lock a spin lock object (**ADVANCED REALTIME THREADS**)

35850 **SYNOPSIS**

35851 THR SPI
```
#include <pthread.h>

int pthread_spin_trylock(pthread_spinlock_t *lock);
```
35852

35853

35854 **DESCRIPTION**

35855 Refer to *pthread_spin_lock*( ).

35856 **NAME**

35857 pthread_spin_unlock — unlock a spin lock object (**ADVANCED REALTIME THREADS**)

35858 **SYNOPSIS**

35859 THR SPI
```
#include <pthread.h>
```

35860
```
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

35861

35862 **DESCRIPTION**

35863 The *pthread_spin_unlock*( ) function shall release the spin lock referenced by *lock* which was
35864 locked via the *pthread_spin_lock*( ) or *pthread_spin_trylock*( ) functions. The results are undefined if
35865 the lock is not held by the calling thread. If there are threads spinning on the lock when
35866 *pthread_spin_unlock*( ) is called, the lock becomes available and an unspecified spinning thread
35867 shall acquire the lock.

35868 The results are undefined if this function is called with an uninitialized thread spin lock.

35869 **RETURN VALUE**

35870 Upon successful completion, the *pthread_spin_unlock*( ) function shall return zero; otherwise, an
35871 error number shall be returned to indicate the error.

35872 **ERRORS**

35873 The *pthread_spin_unlock*( ) function may fail if:

35874 [EINVAL] An invalid argument was specified.

35875 [EPERM] The calling thread does not hold the lock.

35876 This function shall not return an error code of [EINTR].

35877 **EXAMPLES**

35878 None.

35879 **APPLICATION USAGE**

35880 The *pthread_spin_unlock*( ) function is part of the Spin Locks option and need not be provided on
35881 all implementations.

35882 **RATIONALE**

35883 None.

35884 **FUTURE DIRECTIONS**

35885 None.

35886 **SEE ALSO**

35887 *pthread_spin_init*( ), *pthread_spin_destroy*( ), *pthread_spin_lock*( ), *pthread_spin_trylock*( ), the Base
35888 Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

35889 **CHANGE HISTORY**

35890 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

35891 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

35892 **NAME**

35893 pthread_testcancel — set cancelability state

35894 **SYNOPSIS**

35895 THR
```
#include <pthread.h>
```

35896
```
void pthread_testcancel(void);
```

35897

35898 **DESCRIPTION**

35899 Refer to *pthread_setcancelstate*( ).

35900 **NAME**

35901 ptsname — get name of the slave pseudo-terminal device

35902 **SYNOPSIS**

35903 XSI
```
#include <stdlib.h>

char *ptsname(int fildes);
```
35904

35905

35906 **DESCRIPTION**

35907 The *ptsname*( ) function shall return the name of the slave pseudo-terminal device associated
35908 with a master pseudo-terminal device. The *fildes* argument is a file descriptor that refers to the
35909 master device. The *ptsname*( ) function shall return a pointer to a string containing the pathname |
35910 of the corresponding slave device.

35911 The *ptsname*( ) function need not be reentrant. A function that is not required to be reentrant is
35912 not required to be thread-safe.

35913 **RETURN VALUE**

35914 Upon successful completion, *ptsname*( ) shall return a pointer to a string which is the name of the
35915 pseudo-terminal slave device. Upon failure, *ptsname*( ) shall return a null pointer. This could
35916 occur if *fildes* is an invalid file descriptor or if the slave device name does not exist in the file
35917 system.

35918 **ERRORS**

35919 No errors are defined.

35920 **EXAMPLES**

35921 None.

35922 **APPLICATION USAGE**

35923 The value returned may point to a static data area that is overwritten by each call to *ptsname*( ).

35924 **RATIONALE**

35925 None.

35926 **FUTURE DIRECTIONS**

35927 None.

35928 **SEE ALSO**

35929 *grantpt*( ), *open*( ), *ttyname*( ), *unlockpt*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
35930 **<stdlib.h>**

35931 **CHANGE HISTORY**

35932 First released in Issue 4, Version 2.

35933 **Issue 5**

35934 Moved from X/OPEN UNIX extension to BASE.

35935 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

35936 **NAME**

35937 putc — put byte on a stream

35938 **SYNOPSIS**

35939 `#include <stdio.h>`

35940 `int putc(int c, FILE *stream);`

35941 **DESCRIPTION**

35942 CX The functionality described on this reference page is aligned with the ISO C standard. Any
35943 conflict between the requirements described here and the ISO C standard is unintentional. This
35944 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

35945 The *putc*( ) function shall be equivalent to *fputc*( ), except that if it is implemented as a macro it
35946 may evaluate *stream* more than once, so the argument should never be an expression with side
35947 effects.

35948 **RETURN VALUE**

35949 Refer to *fputc*( ).

35950 **ERRORS**

35951 Refer to *fputc*( ).

35952 **EXAMPLES**

35953 None.

35954 **APPLICATION USAGE**

35955 Since it may be implemented as a macro, *putc*( ) may treat a *stream* argument with side effects |
35956 incorrectly. In particular, *putc*(*c*,*f*++) does not necessarily work correctly. Therefore, use of this
35957 function is not recommended in such situations; *fputc*( ) should be used instead.

35958 **RATIONALE**

35959 None.

35960 **FUTURE DIRECTIONS**

35961 None.

35962 **SEE ALSO**

35963 *fputc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

35964 **CHANGE HISTORY**

35965 First released in Issue 1. Derived from Issue 1 of the SVID.

35966 **NAME**

35967 putc_unlocked — stdio with explicit client locking

35968 **SYNOPSIS**

35969 TSF
```
#include <stdio.h>
```

35970
```
int putc_unlocked(int c, FILE *stream);
```

35971

35972 **DESCRIPTION**

35973 Refer to *getc_unlocked*( ).

35974 **NAME**

35975 putchar — put byte on stdout stream

35976 **SYNOPSIS**

35977 `#include <stdio.h>`

35978 `int putchar(int c);`

35979 **DESCRIPTION**

35980 CX The functionality described on this reference page is aligned with the ISO C standard. Any
35981 conflict between the requirements described here and the ISO C standard is unintentional. This
35982 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

35983 The function call *putchar*(*c*) shall be equivalent to *putc*(*c*,*stdout*).

35984 **RETURN VALUE**

35985 Refer to *fputc*( ).

35986 **ERRORS**

35987 Refer to *fputc*( ).

35988 **EXAMPLES**

35989 None.

35990 **APPLICATION USAGE**

35991 None.

35992 **RATIONALE**

35993 None.

35994 **FUTURE DIRECTIONS**

35995 None.

35996 **SEE ALSO**

35997 *putc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

35998 **CHANGE HISTORY**

35999 First released in Issue 1. Derived from Issue 1 of the SVID.

36000 **NAME**

36001 putchar_unlocked — stdio with explicit client locking

36002 **SYNOPSIS**

36003 TSF
```
#include <stdio.h>
```

36004
```
int putchar_unlocked(int c);
```

36005

36006 **DESCRIPTION**

36007 Refer to *getc_unlocked*( ).

NAME

putenv — change or add a value to environment

SYNOPSIS

XSI
```
#include <stdlib.h>

int putenv(char *string);
```

DESCRIPTION

The *putenv*( ) function shall use the *string* argument to set environment variable values. The *string* argument should point to a string of the form "*name=value*. The *putenv*( ) function shall make the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* shall become part of the environment, so altering the string shall change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to *putenv*( ).

The *putenv*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *putenv*( ) shall return 0; otherwise, it shall return a non-zero value and set *errno* to indicate the error.

ERRORS

The *putenv*( ) function may fail if:

[ENOMEM] Insufficient memory was available.

EXAMPLES

Changing the Value of an Environment Variable

The following example changes the value of the *HOME* environment variable to the value **/usr/home**.

```
#include <stdlib.h>
...
static char *var = "HOME=/usr/home";
int ret;

ret = putenv(var);
```

APPLICATION USAGE

The *putenv*( ) function manipulates the environment pointed to by *environ*, and can be used in conjunction with *getenv*( ).

This routine may use *malloc*( ) to enlarge the environment.

A potential error is to call *putenv*( ) with an automatic variable as the argument, then return from the calling function while *string* is still part of the environment.

The *setenv*( ) function is preferred over this function.

RATIONALE

The standard developers noted that *putenv*( ) is the only function available to add to the environment without permitting memory leaks.

36048 **FUTURE DIRECTIONS**

36049 None.

36050 **SEE ALSO**

36051 *exec*, *getenv*( ), *malloc*( ), *setenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

36052 **CHANGE HISTORY**

36053 First released in Issue 1. Derived from Issue 1 of the SVID.

36054 **Issue 5**

36055 The type of the argument to this function is changed from **const char *** to **char ***. This was
36056 indicated as a FUTURE DIRECTION in previous issues.

36057 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

NAME

putmsg, putpmsg — send a message on a STREAM (**STREAMS**)

SYNOPSIS

XSR
```
#include <stropts.h>

int putmsg(int fildes, const struct strbuf *ctlptr,
    const struct strbuf *dataptr, int flags);
int putpmsg(int fildes, const struct strbuf *ctlptr,
    const struct strbuf *dataptr, int band, int flags);
```

DESCRIPTION

The *putmsg*( ) function shall create a message from a process buffer(s) and send the message to a STREAMS file. The message may contain either a data part, a control part, or both. The data and control parts are distinguished by placement in separate buffers, as described below. The semantics of each part are defined by the STREAMS module that receives the message.

The *putpmsg*( ) function is equivalent o *putmsg*( ), except that the process can send messages in different priority bands. Except where noted, all requirements on *putmsg*( ) also pertain to *putpmsg*( ).

The *fildes* argument specifies a file descriptor referencing an open STREAM. The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure.

The *ctlptr* argument points to the structure describing the control part, if any, to be included in the message. The *buf* member in the **strbuf** structure points to the buffer where the control information resides, and the *len* member indicates the number of bytes to be sent. The *maxlen* member is not used by *putmsg*( ). In a similar manner, the argument *dataptr* specifies the data, if any, to be included in the message. The *flags* argument indicates what type of message should be sent and is described further below.

To send the data part of a message, the application shall ensure that *dataptr* is not a null pointer and the *len* member of *dataptr* is 0 or greater. To send the control part of a message, the application shall ensure that the corresponding values are set for *ctlptr*. No data (control) part shall be sent if either *dataptr*(*ctlptr*) is a null pointer or the *len* member of *dataptr*(*ctlptr*) is set to –1.

For *putmsg*( ), if a control part is specified and *flags* is set to RS_HIPRI, a high priority message shall be sent. If no control part is specified, and *flags* is set to RS_HIPRI, *putmsg*( ) shall fail and set *errno* to [EINVAL]. If *flags* is set to 0, a normal message (priority band equal to 0) shall be sent. If a control part and data part are not specified and *flags* is set to 0, no message shall be sent and 0 shall be returned.

For *putpmsg*( ), the flags are different. The *flags* argument is a bitmask with the following mutually-exclusive flags defined: MSG_HIPRI and MSG_BAND. If *flags* is set to 0, *putpmsg*( ) shall fail and set *errno* to [EINVAL]. If a control part is specified and *flags* is set to MSG_HIPRI and *band* is set to 0, a high-priority message shall be sent. If *flags* is set to MSG_HIPRI and either no control part is specified or *band* is set to a non-zero value, *putpmsg*( ) shall fail and set *errno* to [EINVAL]. If *flags* is set to MSG_BAND, then a message shall be sent in the priority band specified by *band*. If a control part and data part are not specified and *flags* is set to MSG_BAND, no message shall be sent and 0 shall be returned.

The *putmsg*( ) function shall block if the STREAM write queue is full due to internal flow control conditions, with the following exceptions:

- For high-priority messages, *putmsg*( ) shall not block on this condition and continues processing the message.

- For other messages, *putmsg*( ) shall not block but shall fail when the write queue is full and O_NONBLOCK is set.

The *putmsg*( ) function shall also block, unless prevented by lack of internal resources, while waiting for the availability of message blocks in the STREAM, regardless of priority or whether O_NONBLOCK has been specified. No partial message shall be sent.

RETURN VALUE

Upon successful completion, *putmsg*( ) and *putpmsg*( ) shall return 0; otherwise, they shall return –1 and set *errno* to indicate the error.

ERRORS

The *putmsg*( ) and *putpmsg*( ) functions shall fail if:

[EAGAIN] A non-priority message was specified, the O_NONBLOCK flag is set, and the STREAM write queue is full due to internal flow control conditions; or buffers could not be allocated for the message that was to be created.

[EBADF] *fildes* is not a valid file descriptor open for writing.

[EINTR] A signal was caught during *putmsg*( ).

[EINVAL] An undefined value is specified in *flags*, or *flags* is set to RS_HIPRI or MSG_HIPRI and no control part is supplied, or the STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer, or *flags* is set to MSG_HIPRI and *band* is non-zero (for *putpmsg*( ) only).

[ENOSR] Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.

[ENOSTR] A STREAM is not associated with *fildes*.

[ENXIO] A hangup condition was generated downstream for the specified STREAM.

[EPIPE] or [EIO] The *fildes* argument refers to a STREAMS-based pipe and the other end of the pipe is closed. A SIGPIPE signal is generated for the calling thread.

[ERANGE] The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAM module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

In addition, *putmsg*( ) and *putpmsg*( ) shall fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of *errno* does not reflect the result of *putmsg*( ) or *putpmsg*( ), but reflects the prior error.

EXAMPLES

Sending a High-Priority Message

The value of *fd* is assumed to refer to an open STREAMS file. This call to *putmsg*() does the following:

1. Creates a high-priority message with a control part and a data part, using the buffers pointed to by *ctrlbuf* and *databuf*, respectively.
2. Sends the message to the STREAMS file identified by *fd*.

```
#include <stropts.h>
#include <string.h>
...
int fd;
char *ctrlbuf = "This is the control part";
char *databuf = "This is the data part";
struct strbuf ctrl;
struct strbuf data;
int ret;

ctrl.buf = ctrlbuf;
ctrl.len = strlen(ctrlbuf);

data.buf = databuf;
data.len = strlen(databuf);

ret = putmsg(fd, &ctrl, &data, MSG_HIPRI);
```

Using putpmsg()

This example has the same effect as the previous example. In this example, however, the *putpmsg*() function creates and sends the message to the STREAMS file.

```
#include <stropts.h>
#include <string.h>
...
int fd;
char *ctrlbuf = "This is the control part";
char *databuf = "This is the data part";
struct strbuf ctrl;
struct strbuf data;
int ret;

ctrl.buf = ctrlbuf;
ctrl.len = strlen(ctrlbuf);

data.buf = databuf;
data.len = strlen(databuf);

ret = putpmsg(fd, &ctrl, &data, 0, MSG_HIPRI);
```

APPLICATION USAGE

None.

36180 **RATIONALE**

36181 None.

36182 **FUTURE DIRECTIONS**

36183 None.

36184 **SEE ALSO**

36185 *getmsg*( ), *poll*( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**,
36186 Section 2.6 (on page 488)

36187 **CHANGE HISTORY**

36188 First released in Issue 4, Version 2.

36189 **Issue 5**

36190 Moved from X/OPEN UNIX extension to BASE.

36191 The following text is removed from the DESCRIPTION: ‘‘The STREAM head guarantees that the
36192 control part of a message generated by *putmsg*( ) is at least 64 bytes in length’’.

36193 **Issue 6**

36194 This function is marked as part of the XSI STREAMS Option Group.

36195 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

putpmsg — send a message on a STREAM (**STREAMS**)

SYNOPSIS

XSR
```
#include <stropts.h>

int putpmsg(int fildes, const struct strbuf *ctlptr,
    const struct strbuf *dataptr, int band, int flags);
```

DESCRIPTION

Refer to *putmsg*( ).

NAME

puts — put a string on standard output

SYNOPSIS

```
#include <stdio.h>

int puts(const char *s);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *puts*( ) function shall write the string pointed to by *s*, followed by a <newline>, to the standard output stream *stdout*. The terminating null byte shall not be written.

CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful execution of *puts*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same stream or a call to *exit*( ) or *abort*( ).

RETURN VALUE

Upon successful completion, *puts*( ) shall return a non-negative number. Otherwise, it shall return EOF, shall set an error indicator for the stream, CX and *errno* shall be set to indicate the error.

ERRORS

Refer to *fputc*( ).

EXAMPLES

Printing to Standard Output

The following example gets the current time, converts it to a string using *localtime*( ) and *asctime*( ), and prints it to standard output using *puts*( ). It then prints the number of minutes to an event for which it is waiting.

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
puts(asctime(localtime(&now)));
printf("There are %d minutes to the event.\n",
    minutes_to_event);
...
```

APPLICATION USAGE

The *puts*( ) function appends a <newline>, while *fputs*( ) does not.

RATIONALE

None.

FUTURE DIRECTIONS

None.

36247 **SEE ALSO**

36248 *fopen*( ), *fputs*( ), *putc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

36249 **CHANGE HISTORY**

36250 First released in Issue 1. Derived from Issue 1 of the SVID.

36251 **Issue 6**

36252 Extensions beyond the ISO C standard are now marked.

36253 **NAME**

36254 pututxline — put an entry into user accounting database

36255 **SYNOPSIS**

36256 XSI
```
#include <utmpx.h>
```

36257
```
struct utmpx *pututxline(const struct utmpx *utmpx);
```

36258

36259 **DESCRIPTION**

36260 Refer to *endutxent*( ).

NAME

putwc — put a wide character on a stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t putwc(wchar_t wc, FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *putwc*( ) function shall be equivalent to *fputwc*( ), except that if it is implemented as a macro it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

RETURN VALUE

Refer to *fputwc*( ).

ERRORS

Refer to *fputwc*( ).

EXAMPLES

None.

APPLICATION USAGE

Since it may be implemented as a macro, *putwc*( ) may treat a *stream* argument with side effects incorrectly. In particular, *putwc*(*wc*,*f*++) need not work correctly. Therefore, use of this function is not recommended; *fputwc*( ) should be used instead.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fputwc( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released as a World-wide Portability Interface in Issue 4.

Issue 5

Aligned with ISO/IEC 9899: 1990/Amendment 1: 1995 (E). Specifically, the type of argument *wc* is changed from **wint_t** to **wchar_t**.

The Optional Header (OH) marking is removed from **<stdio.h>**.

36296 **NAME**

36297 putwchar — put a wide character on stdout stream

36298 **SYNOPSIS**

36299
```
#include <wchar.h>
```

36300
```
wint_t putwchar(wchar_t wc);
```

36301 **DESCRIPTION**

36302 CX The functionality described on this reference page is aligned with the ISO C standard. Any
36303 conflict between the requirements described here and the ISO C standard is unintentional. This
36304 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

36305 The function call *putwchar*(*wc*) shall be equivalent to *putwc*(*wc*,*stdout*).

36306 **RETURN VALUE**

36307 Refer to *fputwc*( ).

36308 **ERRORS**

36309 Refer to *fputwc*( ).

36310 **EXAMPLES**

36311 None.

36312 **APPLICATION USAGE**

36313 None.

36314 **RATIONALE**

36315 None.

36316 **FUTURE DIRECTIONS**

36317 None.

36318 **SEE ALSO**

36319 *fputwc*( ), *putwc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

36320 **CHANGE HISTORY**

36321 First released in Issue 4.

36322 **Issue 5**

36323 Aligned with ISO/IEC 9899: 1990/Amendment 1: 1995 (E). Specifically, the type of argument *wc*
36324 is changed from **wint_t** to **wchar_t**.

36325 **NAME**

36326 pwrite — write on a file

36327 **SYNOPSIS**

36328 `#include <unistd.h>`

36329 XSI `ssize_t pwrite(int fildes, const void *buf, size_t nbyte,`
36330 `    off_t offset);`

36331

36332 **DESCRIPTION**

36333 Refer to *write*( ).

36334 **NAME**

36335 qsort — sort a table of data

36336 **SYNOPSIS**

36337
```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
    int (*compar)(const void *, const void *));
```

36340 **DESCRIPTION**

36341 CX The functionality described on this reference page is aligned with the ISO C standard. Any
36342 conflict between the requirements described here and the ISO C standard is unintentional. This
36343 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

36344 The *qsort*( ) function shall sort an array of *nel* objects, the initial element of which is pointed to by
36345 *base*. The size of each object, in bytes, is specified by the *width* argument.

36346 The contents of the array shall be sorted in ascending order according to a comparison function.
36347 The *compar* argument is a pointer to the comparison function, which is called with two
36348 arguments that point to the elements being compared. The application shall ensure that the
36349 function returns an integer less than, equal to, or greater than 0, if the first argument is
36350 considered respectively less than, equal to, or greater than the second. If two members compare
36351 as equal, their order in the sorted array is unspecified.

36352 **RETURN VALUE**

36353 The *qsort*( ) function shall not return a value.

36354 **ERRORS**

36355 No errors are defined.

36356 **EXAMPLES**

36357 None.

36358 **APPLICATION USAGE**

36359 The comparison function need not compare every byte, so arbitrary data may be contained in
36360 the elements in addition to the values being compared.

36361 **RATIONALE**

36362 None.

36363 **FUTURE DIRECTIONS**

36364 None.

36365 **SEE ALSO**

36366 The Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

36367 **CHANGE HISTORY**

36368 First released in Issue 1. Derived from Issue 1 of the SVID.

36369 **Issue 6**

36370 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

raise — send a signal to the executing process

SYNOPSIS

```
#include <signal.h>

int raise(int sig);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *raise*( ) function shall send the signal *sig* to the executing thread or process. If a signal handler is called, the *raise*( ) function shall not return until after the signal handler does.

THR If the implementation supports the Threads option, the effect of the *raise*( ) function shall be equivalent to calling:

```
pthread_kill(pthread_self(), sig);
```

CX Otherwise, the effect of the *raise*( ) function shall be equivalent to calling:

```
kill(getpid(), sig);
```

RETURN VALUE

CX Upon successful completion, 0 shall be returned. Otherwise, a non-zero value shall be returned and *errno* shall be set to indicate the error.

ERRORS

The *raise*( ) function shall fail if:

CX [EINVAL] The value of the *sig* argument is an invalid signal number.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The term ‘‘thread’’ is an extension to the ISO C standard.

FUTURE DIRECTIONS

None.

SEE ALSO

kill( ), *sigaction*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EINVAL] error condition is added.

NAME

rand, rand_r, srand — pseudo-random number generator

SYNOPSIS

```
#include <stdlib.h>

int rand(void);
int rand_r(unsigned *seed);
void srand(unsigned seed);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *rand*( ) function shall compute a sequence of pseudo-random integers in the range 0 to XSI {RAND_MAX} with a period of at least 2^{32}.

CX The *rand*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *rand_r*( ) function shall compute a sequence of pseudo-random integers in the range 0 to {RAND_MAX}. (The value of the {RAND_MAX} macro shall be at least 32 767.)

If *rand_r*( ) is called with the same initial value for the object pointed to by *seed* and that object is not modified between successive returns and calls to *rand_r*( ), the same sequence shall be generated.

The *srand*( ) function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to *rand*( ). If *srand*( ) is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If *rand*( ) is called before any calls to *srand*( ) are made, the same sequence shall be generated as when *srand*( ) is first called with a seed value of 1.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *rand*( ) or *srand*( ).

RETURN VALUE

The *rand*( ) function shall return the next pseudo-random number in the sequence.

TSF The *rand_r*( ) function shall return a pseudo-random integer.

The *srand*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

Generating a Pseudo-Random Number Sequence

The following example demonstrates how to generate a sequence of pseudo-random numbers.

```
#include <stdio.h>
#include <stdlib.h>
...
    long count, i;
    char *keystr;
    int elementlen, len;
    char c;
```

```
...
/* Initial random number generator. */
    srand(1);

    /* Create keys using only lower case characters */
    len = 0;
    for (i=0; i<count; i++) {
        while (len < elementlen) {
            c = (char) (rand() % 128);
            if (islower(c))
                keystr[len++] = c;
        }

        keystr[len] = '\0';
        printf("%s Element%0*ld\n", keystr, elementlen, i);
        len = 0;
    }
```

Generating the Same Sequence on Different Machines

The following code defines a pair of functions that could be incorporated into applications wishing to ensure that the same sequence of numbers is generated across different machines.

```
static unsigned long next = 1;
int myrand(void)  /* RAND_MAX assumed to be 32767. */
{
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}

void mysrand(unsigned seed)
{
    next = seed;
}
```

APPLICATION USAGE

The *drand48*( ) function provides a much more elaborate random number generator.

RATIONALE

The ISO C standard *rand*( ) and *srand*( ) functions allow per-process pseudo-random streams shared by all threads. Those two functions need not change, but there has to be mutual-exclusion that prevents interference between two threads concurrently accessing the random number generator.

With regard to *rand*( ), there are two different behaviors that may be wanted in a multi-threaded program:

1. A single per-process sequence of pseudo-random numbers that is shared by all threads that call *rand*( )
2. A different sequence of pseudo-random numbers for each thread that calls *rand*( )

This is provided by the modified thread-safe function based on whether the seed value is global to the entire process or local to each thread.

This does not address the known deficiencies of the *rand*( ) function implementations, which have been approached by maintaining more state. In effect, this specifies new thread-safe forms of a deficient function.

36504 **FUTURE DIRECTIONS**

36505 None.

36506 **SEE ALSO**

36507 *drand48*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

36508 **CHANGE HISTORY**

36509 First released in Issue 1. Derived from Issue 1 of the SVID.

36510 **Issue 5**

36511 The *rand_r*( ) function is included for alignment with the POSIX Threads Extension.

36512 A note indicating that the *rand*( ) function need not be reentrant is added to the DESCRIPTION.

36513 **Issue 6**

36514 Extensions beyond the ISO C standard are now marked.

36515 The *rand_r*( ) function is marked as part of the Thread-Safe Functions option.

36516 **NAME**

36517 random — generate pseudo-random number

36518 **SYNOPSIS**

36519 XSI
```
#include <stdlib.h>
```
36520
```
long random(void);
```
36521

36522 **DESCRIPTION**

36523 Refer to *initstate*( ).

36524 **NAME**

36525 pread, read — read from a file

36526 **SYNOPSIS**

36527 `#include <unistd.h>`

36528 XSI `ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);`

36529 `ssize_t read(int fildes, void *buf, size_t nbyte);`

36530 **DESCRIPTION**

36531 The *read*( ) function shall attempt to read *nbyte* bytes from the file associated with the open file
36532 descriptor, *fildes*, into the buffer pointed to by *buf*. The behavior of multiple concurrent reads on
36533 the same pipe, FIFO, or terminal device is unspecified.

36534 Before any action described below is taken, and if *nbyte* is zero, the *read*( ) function may detect
36535 and return errors as described below. In the absence of errors, or if error detection is not
36536 performed, the *read*( ) function shall return zero and have no other results.

36537 On files that support seeking (for example, a regular file), the *read*( ) shall start at a position in
36538 the file given by the file offset associated with *fildes*. The file offset shall be incremented by the
36539 number of bytes actually read.

36540 Files that do not support seeking—for example, terminals—always read from the current
36541 position. The value of a file offset associated with such a file is undefined.

36542 No data transfer shall occur past the current end-of-file. If the starting position is at or after the
36543 end-of-file, 0 shall be returned. If the file refers to a device special file, the result of subsequent
36544 *read*( ) requests is implementation-defined.

36545 If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation-defined.

36546 When attempting to read from an empty pipe or FIFO:

36547 - If no process has the pipe open for writing, *read*( ) shall return 0 to indicate end-of-file.

36548 - If some process has the pipe open for writing and O_NONBLOCK is set, *read*( ) shall return
36549 –1 and set *errno* to [EAGAIN].

36550 - If some process has the pipe open for writing and O_NONBLOCK is clear, *read*( ) shall block
36551 the calling thread until some data is written or the pipe is closed by all processes that had the
36552 pipe open for writing.

36553 When attempting to read a file (other than a pipe or FIFO) that supports non-blocking reads and
36554 has no data currently available:

36555 - If O_NONBLOCK is set, *read*( ) shall return –1 and set *errno* to [EAGAIN].

36556 - If O_NONBLOCK is clear, *read*( ) shall block the calling thread until some data becomes
36557 available.

36558 - The use of the O_NONBLOCK flag has no effect if there is some data available.

36559 The *read*( ) function reads data previously written to a file. If any portion of a regular file prior to
36560 the end-of-file has not been written, *read*( ) shall return bytes with value 0. For example, *lseek*( )
36561 allows the file offset to be set beyond the end of existing data in the file. If data is later written at
36562 this point, subsequent reads in the gap between the previous end of data and the newly written
36563 data shall return bytes with value 0 until data is written into the gap.

36564 Upon successful completion, where *nbyte* is greater than 0, *read*( ) shall mark for update the
36565 *st_atime* field of the file, and shall return the number of bytes read. This number shall never be
36566 greater than *nbyte*. The value returned may be less than *nbyte* if the number of bytes left in the

file is less than *nbyte*, if the *read*( ) request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than *nbyte* bytes immediately available for reading. For example, a *read*( ) from a file associated with a terminal may return one typed line of data.

If a *read*( ) is interrupted by a signal before it reads any data, it shall return –1 with *errno* set to [EINTR].

If a *read*( ) is interrupted by a signal after it has successfully read some data, it shall return the number of bytes read.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with *fildes*.

If *fildes* refers to a socket, *read*( ) shall be equivalent to *recv*( ) with no flags set.

SIO If the O_DSYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion. If the O_SYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

SHM If *fildes* refers to a shared memory object, the result of the *read*( ) function is unspecified.

TYM If *fildes* refers to a typed memory object, the result of the *read*( ) function is unspecified.

XSR A *read*( ) from a STREAMS file can read data in three different modes: *byte-stream* mode, *message-nondiscard* mode, and *message-discard* mode. The default shall be byte-stream mode. This can be changed using the I_SRDOPT *ioctl*( ) request, and can be tested with the I_GRDOPT *ioctl*( ). In byte-stream mode, *read*( ) shall retrieve data from the STREAM until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, *read*( ) shall retrieve data until as many bytes as were requested are transferred, or until a message boundary is reached. If *read*( ) does not retrieve all the data in a message, the remaining data shall be left on the STREAM, and can be retrieved by the next *read*( ) call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the *read*( ) returns shall be discarded, and shall not be available for a subsequent *read*( ), *getmsg*( ), or *getpmsg*( ) call.

How *read*( ) handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, *read*( ) shall accept data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The *read*( ) function shall then return the number of bytes read, and place the zero-byte message back on the STREAM to be retrieved by the next *read*( ), *getmsg*( ), or *getpmsg*( ). In message-nondiscard mode or message-discard mode, a zero-byte message shall return 0 and the message shall be removed from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message shall be removed from the STREAM and 0 shall be returned, regardless of the read mode.

A *read*( ) from a STREAMS file shall return the data in the message at the front of the STREAM head read queue, regardless of the priority band of the message.

By default, STREAMs are in control-normal mode, in which a *read*( ) from a STREAMS file can only process messages that contain a data part but do not contain a control part. The *read*( ) shall fail if a message containing a control part is encountered at the STREAM head. This default action can be changed by placing the STREAM in either control-data mode or control-discard mode with the I_SRDOPT *ioctl*( ) command. In control-data mode, *read*( ) shall convert any control part to data and pass it to the application before passing any data part originally present

in the same message. In control-discard mode, *read*( ) shall discard message control parts but return to the process any data part in the message.

In addition, *read*( ) shall fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of *errno* shall not reflect the result of *read*( ), but reflects the prior error. If a hangup occurs on the STREAM being read, *read*( ) shall continue to operate normally until the STREAM head read queue is empty. Thereafter, it shall return 0.

XSI The *pread*( ) function shall be equivalent to *read*( ), except that it shall read from a given position in the file without changing the file pointer. The first three arguments to *pread*( ) are the same as *read*( ) with the addition of a fourth argument offset for the desired position inside the file. An attempt to perform a *pread*( ) on a file that is incapable of seeking shall result in an error.

RETURN VALUE

XSI Upon successful completion, *read*( ) and *pread*( ) shall return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions shall return –1 and set *errno* to indicate the error.

ERRORS

XSI The *read*( ) and *pread*( ) functions shall fail if:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the process would be delayed.

[EBADF] The *fildes* argument is not a valid file descriptor open for reading.

XSR [EBADMSG] The file is a STREAM file that is set to control-normal mode and the message waiting to be read includes a control part.

[EINTR] The read operation was terminated due to the receipt of a signal, and no data was transferred.

XSR [EINVAL] The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

[EIO] The process is a member of a background process attempting to read from its controlling terminal, the process is ignoring or blocking the SIGTTIN signal, or the process group is orphaned. This error may also be generated for implementation-defined reasons.

XSI [EISDIR] The *fildes* argument refers to a directory and the implementation does not allow the directory to be read using *read*( ) or *pread*( ). The *readdir*( ) function should be used instead.

[EOVERFLOW] The file is a regular file, *nbyte* is greater than 0, the starting position is before the end-of-file, and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildes*.

The *read*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]
The file descriptor is for a socket, is marked O_NONBLOCK, and no data is waiting to be received.

[ECONNRESET] A read was attempted on a socket and the connection was forcibly closed by its peer.

[ENOTCONN] A read was attempted on a socket that is not connected.

[ETIMEDOUT] A read was attempted on a socket and a transmission timeout occurred.

36656 XSI The *read*( ) and *pread*( ) functions may fail if:

36657 [EIO] A physical I/O error has occurred.

36658 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

36659 [ENOMEM] Insufficient memory was available to fulfill the request.

36660 [ENXIO] A request was made of a nonexistent device, or the request was outside the
36661 capabilities of the device.

36662 The *pread*( ) function shall fail, and the file pointer shall remain unchanged, if:

36663 XSI [EINVAL] The *offset* argument is invalid. The value is negative.

36664 XSI [EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the
36665 offset maximum associated with the file.

36666 XSI [ENXIO] A request was outside the capabilities of the device.

36667 XSI [ESPIPE] *fildes* is associated with a pipe or FIFO.

36668 **EXAMPLES**

36669 **Reading Data into a Buffer**

36670 The following example reads data from the file associated with the file descriptor *fd* into the
36671 buffer pointed to by *buf*.

```
36672 #include <sys/types.h>
36673 #include <unistd.h>
36674 ...
36675 char buf[20];
36676 size_t nbytes;
36677 ssize_t bytes_read;
36678 int fd;
36679 ...
36680 nbytes = sizeof(buf);
36681 bytes_read = read(fd, buf, nbytes);
36682 ...
```

36683 **APPLICATION USAGE**

36684 None.

36685 **RATIONALE**

36686 This volume of IEEE Std 1003.1-200x does not specify the value of the file offset after an error is
36687 returned; there are too many cases. For programming errors, such as [EBADF], the concept is
36688 meaningless since no file is involved. For errors that are detected immediately, such as
36689 [EAGAIN], clearly the pointer should not change. After an interrupt or hardware error, however,
36690 an updated value would be very useful and is the behavior of many implementations.

36691 Note that a *read*( ) of zero bytes does not modify *st_atime*. A *read*( ) that requests more than zero
36692 bytes, but returns zero, shall modify *st_atime*.

36693 Implementations are allowed, but not required, to perform error checking for *read*( ) requests of
36694 zero bytes.

Input and Output

The use of I/O with large byte counts has always presented problems. Ideas such as *lread*() and *lwrite*() (using and returning **long**s) were considered at one time. The current solution is to use abstract types on the ISO C standard function to *read*() and *write*(). The abstract types can be declared so that existing functions work, but can also be declared so that larger types can be represented in future implementations. It is presumed that whatever constraints limit the maximum range of **size_t** also limit portable I/O requests to the same range. This volume of IEEE Std 1003.1-200x also limits the range further by requiring that the byte count be limited so that a signed return value remains meaningful. Since the return type is also a (signed) abstract type, the byte count can be defined by the implementation to be larger than an **int** can hold.

The standard developers considered adding atomicity requirements to a pipe or FIFO, but recognized that due to the nature of pipes and FIFOs there could be no guarantee of atomicity of reads of {PIPE_BUF} or any other size that would be an aid to applications portability.

This volume of IEEE Std 1003.1-200x requires that no action be taken for *read*() or *write*() when *nbyte* is zero. This is not intended to take precedence over detection of errors (such as invalid buffer pointers or file descriptors). This is consistent with the rest of this volume of IEEE Std 1003.1-200x, but the phrasing here could be misread to require detection of the zero case before any other errors. A value of zero is to be considered a correct value, for which the semantics are a no-op.

I/O is intended to be atomic to ordinary files and pipes and FIFOs. Atomic means that all the bytes from a single operation that started out together end up together, without interleaving from other I/O operations. It is a known attribute of terminals that this is not honored, and terminals are explicitly (and implicitly permanently) excepted, making the behavior unspecified. The behavior for other device types is also left unspecified, but the wording is intended to imply that future standards might choose to specify atomicity (or not).

There were recommendations to add format parameters to *read*() and *write*() in order to handle networked transfers among heterogeneous file system and base hardware types. Such a facility may be required for support by the OSI presentation of layer services. However, it was determined that this should correspond with similar C-language facilities, and that is beyond the scope of this volume of IEEE Std 1003.1-200x. The concept was suggested to the developers of the ISO C standard for their consideration as a possible area for future work.

In 4.3 BSD, a *read*() or *write*() that is interrupted by a signal before transferring any data does not by default return an [EINTR] error, but is restarted. In 4.2 BSD, 4.3 BSD, and the Eighth Edition, there is an additional function, *select*(), whose purpose is to pause until specified activity (data to read, space to write, and so on) is detected on specified file descriptors. It is common in applications written for those systems for *select*() to be used before *read*() in situations (such as keyboard input) where interruption of I/O due to a signal is desired.

The issue of which files or file types are interruptible is considered an implementation design issue. This is often affected primarily by hardware and reliability issues.

There are no references to actions taken following an ‘‘unrecoverable error’’. It is considered beyond the scope of this volume of IEEE Std 1003.1-200x to describe what happens in the case of hardware errors.

Previous versions of IEEE Std 1003.1-200x allowed two very different behaviors with regard to the handling of interrupts. In order to minimize the resulting confusion, it was decided that IEEE Std 1003.1-200x should support only one of these behaviors. Historical practice on AT&T-derived systems was to have *read*() and *write*() return –1 and set *errno* to [EINTR] when interrupted after some, but not all, of the data requested had been transferred. However, the U.S. Department of Commerce FIPS 151-1 and FIPS 151-2 require the historical BSD behavior, in

which *read*( ) and *write*( ) return the number of bytes actually transferred before the interrupt. If –1 is returned when any data is transferred, it is difficult to recover from the error on a seekable device and impossible on a non-seekable device. Most new implementations support this behavior. The behavior required by IEEE Std 1003.1-200x is to return the number of bytes transferred.

IEEE Std 1003.1-200x does not specify when an implementation that buffers *read*( )s actually moves the data into the user-supplied buffer, so an implementation may chose to do this at the latest possible moment. Therefore, an interrupt arriving earlier may not cause *read*( ) to return a partial byte count, but rather to return –1 and set *errno* to [EINTR].

Consideration was also given to combining the two previous options, and setting *errno* to [EINTR] while returning a short count. However, not only is there no existing practice that implements this, it is also contradictory to the idea that when *errno* is set, the function responsible shall return –1.

FUTURE DIRECTIONS

None.

SEE ALSO

fcntl( ), *ioctl*( ), *lseek*( ), *open*( ), *pipe*( ), *readv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**, **<sys/uio.h>**, **<unistd.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

The *pread*( ) function is added.

Issue 6

The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are marked as part of the XSI STREAMS Option Group.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION now states that if *read*( ) is interrupted by a signal after it has successfully read some data, it returns the number of bytes read. In Issue 3, it was optional whether *read*( ) returned the number of bytes read, or whether it returned –1 with *errno* set to [EINTR]. This is a FIPS requirement.
- In the DESCRIPTION, text is added to indicate that for regular files, no data transfer occurs past the offset maximum established in the open file description associated with *fildes*. This change is to support large files.
- The [EOVERFLOW] mandatory error condition is added.
- The [ENXIO] optional error condition is added.

Text referring to sockets is added to the DESCRIPTION.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effect of reading zero bytes is clarified.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that *read*( ) results are unspecified for typed memory objects.

New RATIONALE is added to explain the atomicity requirements for input and output operations.

The following error conditions are added for operations on sockets: [EAGAIN], [ECONNRESET], [ENOTCONN], and [ETIMEDOUT].

The [EIO] error is changed to ‘‘may fail’’.

The following error conditions are added for operations on sockets: [ENOBUFS] and [ENOMEM]. |

The *readv*( ) function is split out into a separate reference page. |

NAME

readdir, readdir_r — read directory

SYNOPSIS

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
TSF int readdir_r(DIR *restrict dirp, struct dirent *restrict entry,
        struct dirent **restrict result);
```

DESCRIPTION

The type **DIR**, which is defined in the **<dirent.h>** header, represents a *directory stream*, which is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files; files may be removed from a directory or added to a directory asynchronously to the operation of *readdir*( ).

The *readdir*( ) function shall return a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument *dirp*, and position the directory stream at the next entry. It shall return a null pointer upon reaching the end of the directory stream. The structure **dirent** defined in the **<dirent.h>** header describes a directory entry.

The *readdir*( ) function shall not return directory entries containing empty names. If entries for dot or dot-dot exist, one entry shall be returned for dot and one entry shall be returned for dot-dot; otherwise, they shall not be returned.

The pointer returned by *readdir*( ) points to data which may be overwritten by another call to *readdir*( ) on the same directory stream. This data is not overwritten by another call to *readdir*( ) on a different directory stream.

If a file is removed from or added to the directory after the most recent call to *opendir*( ) or *rewinddir*( ), whether a subsequent call to *readdir*( ) returns an entry for that file is unspecified.

The *readdir*( ) function may buffer several directory entries per actual read operation; *readdir*( ) shall mark for update the *st_atime* field of the directory each time the directory is actually read.

After a call to *fork*( ), either the parent or child (but not both) may continue processing the directory stream using *readdir*( ), *rewinddir*( ), XSI or *seekdir*( ). If both the parent and child processes use these functions, the result is undefined.

If the entry names a symbolic link, the value of the *d_ino* member is unspecified.

The *readdir*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *readdir_r*( ) function shall initialize the **dirent** structure referenced by *entry* to represent the directory entry at the current position in the directory stream referred to by *dirp*, store a pointer to this structure at the location referenced by *result*, and position the directory stream at the next entry.

The storage pointed to by *entry* shall be large enough for a **dirent** with an array of **char** *d_name* members containing at least {NAME_MAX} plus one elements.

Upon successful return, the pointer returned at **result* shall have the same value as the argument *entry*. Upon reaching the end of the directory stream, this pointer shall have the value NULL.

The *readdir_r*( ) function shall not return directory entries containing empty names.

If a file is removed from or added to the directory after the most recent call to *opendir*() or *rewinddir*(), whether a subsequent call to *readdir_r*() returns an entry for that file is unspecified.

The *readdir_r*() function may buffer several directory entries per actual read operation; the *readdir_r*() function shall mark for update the *st_atime* field of the directory each time the directory is actually read.

Applications wishing to check for error situations should set *errno* to 0 before calling *readdir*(). If *errno* is set to non-zero on return, an error occurred.

RETURN VALUE

Upon successful completion, *readdir*() shall return a pointer to an object of type **struct dirent**. When an error is encountered, a null pointer shall be returned and *errno* shall be set to indicate the error. When the end of the directory is encountered, a null pointer shall be returned and *errno* is not changed.

TSF If successful, the *readdir_r*() function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *readdir*() function shall fail if:

[EOVERFLOW] One of the values in the structure to be returned cannot be represented correctly.

The *readdir*() function may fail if:

[EBADF] The *dirp* argument does not refer to an open directory stream.

[ENOENT] The current position of the directory stream is invalid.

The *readdir_r*() function may fail if:

[EBADF] The *dirp* argument does not refer to an open directory stream.

EXAMPLES

The following sample code searches the current directory for the entry *name*:

```
dirp = opendir(".");

while (dirp) {
    errno = 0;
    if ((dp = readdir(dirp)) != NULL) {
        if (strcmp(dp->d_name, name) == 0) {
            closedir(dirp);
            return FOUND;
        }
    } else {
        if (errno == 0) {
            closedir(dirp);
            return NOT_FOUND;
        }
        closedir(dirp);
        return READ_ERROR;
    }
}

return OPEN_ERROR;
```

36882 **APPLICATION USAGE**

36883 The *readdir*( ) function should be used in conjunction with *opendir*( ), *closedir*( ), and *rewinddir*( ) to
36884 examine the contents of the directory.

36885 The *readdir_r*( ) function is thread-safe and shall return values in a user-supplied buffer instead
36886 of possibly using a static data area that may be overwritten by each call.

36887 **RATIONALE**

36888 The returned value of *readdir*( ) merely *represents* a directory entry. No equivalence should be
36889 inferred.

36890 Historical implementations of *readdir*( ) obtain multiple directory entries on a single read
36891 operation, which permits subsequent *readdir*( ) operations to operate from the buffered
36892 information. Any wording that required each successful *readdir*( ) operation to mark the
36893 directory *st_atime* field for update would militate against the historical performance-oriented
36894 implementations.

36895 Since *readdir*( ) returns NULL when it detects an error and when the end of the directory is
36896 encountered, an application that needs to tell the difference must set *errno* to zero before the call
36897 and check it if NULL is returned. Since the function must not change *errno* in the second case |
36898 and must set it to a non-zero value in the first case, a zero *errno* after a call returning NULL
36899 indicates end of directory; otherwise, an error.

36900 Routines to deal with this problem more directly were proposed:

36901 ```
int derror (dirp)
```
36902 ```
DIR *dirp;
```

36903 ```
void clearderr (dirp)
```
36904 ```
DIR *dirp;
```

36905 The first would indicate whether an error had occurred, and the second would clear the error
36906 indication. The simpler method involving *errno* was adopted instead by requiring that *readdir*( )
36907 not change *errno* when end-of-directory is encountered.

36908 An error or signal indicating that a directory has changed while open was considered but
36909 rejected.

36910 The thread-safe version of the directory reading function returns values in a user-supplied buffer
36911 instead of possibly using a static data area that may be overwritten by each call. Either the
36912 {NAME_MAX} compile-time constant or the corresponding *pathconf*( ) option can be used to
36913 determine the maximum sizes of returned pathnames. |

36914 **FUTURE DIRECTIONS**

36915 None.

36916 **SEE ALSO**

36917 *closedir*( ), *lstat*( ), *opendir*( ), *rewinddir*( ), *symlink*( ), the Base Definitions volume of
36918 IEEE Std 1003.1-200x, **<dirent.h>**, **<sys/types.h>**

36919 **CHANGE HISTORY**

36920 First released in Issue 2.

36921 **Issue 5**

36922 Large File Summit extensions are added.

36923 The *readdir_r*( ) function is included for alignment with the POSIX Threads Extension.

36924 A note indicating that the *readdir*( ) function need not be reentrant is added to the
36925 DESCRIPTION.

Issue 6

The *readdir_r*( ) function is marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U026/7 is applied, correcting the prototype for *readdir_r*( ).

The Open Group Corrigendum U026/8 is applied, clarifying the wording of the successful return for the *readdir_r*( ) function.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- A statement is added to the DESCRIPTION indicating the disposition of certain fields in **struct dirent** when an entry refers to a symbolic link.
- The [EOVERFLOW] mandatory error condition is added. This change is to support large files.
- The [ENOENT] optional error condition is added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The **restrict** keyword is added to the *readdir_r*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

36945 **NAME**

36946 readlink — read the contents of a symbolic link

36947 **SYNOPSIS**

36948 ```
#include <unistd.h>
```

36949 ```
ssize_t readlink(const char *restrict path, char *restrict buf,
36950     size_t bufsize);
```

36951 **DESCRIPTION**

36952 The *readlink*( ) function shall place the contents of the symbolic link referred to by *path* in the
36953 buffer *buf* which has size *bufsize*. If the number of bytes in the symbolic link is less than *bufsize*,
36954 the contents of the remainder of *buf* are unspecified. If the *buf* argument is not large enough to
36955 contain the link content, the first *bufsize* bytes shall be placed in *buf*.

36956 If the value of *bufsize* is greater than {SSIZE_MAX}, the result is implementation-defined.

36957 **RETURN VALUE**

36958 Upon successful completion, *readlink*( ) shall return the count of bytes placed in the buffer.
36959 Otherwise, it shall return a value of –1, leave the buffer unchanged, and set *errno* to indicate the
36960 error.

36961 **ERRORS**

36962 The *readlink*( ) function shall fail if:

36963 [EACCES] Search permission is denied for a component of the path prefix of *path*.

36964 [EINVAL] The *path* argument names a file that is not a symbolic link.

36965 [EIO] An I/O error occurred while reading from the file system.

36966 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
36967 argument.

36968 [ENAMETOOLONG]
36969 The length of the *path* argument exceeds {PATH_MAX} or a pathname |
36970 component is longer than {NAME_MAX}. |

36971 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

36972 [ENOTDIR] A component of the path prefix is not a directory.

36973 The *readlink*( ) function may fail if:

36974 [EACCES] Read permission is denied for the directory.

36975 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
36976 resolution of the *path* argument.

36977 [ENAMETOOLONG]
36978 As a result of encountering a symbolic link in resolution of the *path* argument, |
36979 the length of the substituted pathname string exceeded {PATH_MAX}. |

EXAMPLES

Reading the Name of a Symbolic Link

The following example shows how to read the name of a symbolic link named **/modules/pass1**.

```
#include <unistd.h>

char buf[1024];
int len;
...
if ((len = readlink("/modules/pass1", buf, sizeof(buf)-1)) != -1);
buf[len] = '\0';
```

APPLICATION USAGE

Conforming applications should not assume that the returned contents of the symbolic link are null-terminated.

RATIONALE

Since IEEE Std 1003.1-200x does not require any association of file times with symbolic links, there is no requirement that file times be updated by *readlink*(). The type associated with *bufsiz* is a **size_t** in order to be consistent with both the ISO C standard and the definition of *read*(). The behavior specified for *readlink*() when *bufsiz* is zero represents historical practice. For this case, the standard developers considered a change whereby *readlink*() would return the number of non-null bytes contained in the symbolic link with the buffer *buf* remaining unchanged; however, since the **stat** structure member *st_size* value can be used to determine the size of buffer necessary to contain the contents of the symbolic link as returned by *readlink*(), this proposal was rejected, and the historical practice retained.

FUTURE DIRECTIONS

None.

SEE ALSO

lstat(), *stat*(), *symlink*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The return type is changed to **ssize_t**, to align with the IEEE P1003.1a draft standard.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- This function is made mandatory.
- In this function it is possible for the return value to exceed the range of the type **ssize_t** (since **size_t** has a larger range of positive values than **ssize_t**). A sentence restricting the size of the **size_t** object is added to the description to resolve this conflict.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The FUTURE DIRECTIONS section is changed to None.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

The **restrict** keyword is added to the *readlink*() prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

readv — read a vector

SYNOPSIS

XSI

```
#include <sys/uio.h>

ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);
```

DESCRIPTION

The *readv*( ) function shall be equivalent to *read*( ), except as described below. The *readv*( ) function shall place the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*−1]. The *iovcnt* argument is valid if greater than 0 and less than or equal to {IOV_MAX}.

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. The *readv*( ) function shall always fill an area completely before proceeding to the next.

Upon successful completion, *readv*( ) shall mark for update the *st_atime* field of the file.

RETURN VALUE

Refer to *read*( ).

ERRORS

Refer to *read*( ).

In addition, the *readv*( ) function shall fail if:

[EINVAL] The sum of the *iov_len* values in the *iov* array overflowed an **ssize_t**.

The *readv*( ) function may fail if:

[EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than {IOV_MAX}.

EXAMPLES

Reading Data into an Array

The following example reads data from the file associated with the file descriptor *fd* into the buffers specified by members of the *iov* array.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
...
ssize_t bytes_read;
int fd;
char buf0[20];
char buf1[30];
char buf2[40];
int iovcnt;
struct iovec iov[3];

iov[0].iov_base = buf0;
iov[0].iov_len = sizeof(buf0);
iov[1].iov_base = buf1;
iov[1].iov_len = sizeof(buf1);
iov[2].iov_base = buf2;
iov[2].iov_len = sizeof(buf2);
```

```
...
iovcnt = sizeof(iov) / sizeof(struct iovec);

bytes_read = readv(fd, iov, iovcnt);
...
```

APPLICATION USAGE

None.

RATIONALE

Refer to *read*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

read( ), *writev*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/uio.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 6

Split out from the *read*( ) reference page.

NAME

realloc — memory reallocator

SYNOPSIS

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *realloc*( ) function shall change the size of the memory object pointed to by *ptr* to the size specified by *size*. The contents of the object shall remain unchanged up to the lesser of the new and old sizes. If the new size of the memory object would require movement of the object, the space for the previous instantiation of the object is freed. If the new size is larger, the contents of the newly allocated portion of the object are unspecified. If *size* is 0 and *ptr* is not a null pointer, the object pointed to is freed. If the space cannot be allocated, the object shall remain unchanged.

If *ptr* is a null pointer, *realloc*( ) shall be equivalent to *malloc*( ) for the specified size.

If *ptr* does not match a pointer returned earlier by *calloc*( ), *malloc*( ), or *realloc*( ) or if the space has previously been deallocated by a call to *free*( ) or *realloc*( ), the behavior is undefined.

The order and contiguity of storage allocated by successive calls to *realloc*( ) is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned shall point to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned.

RETURN VALUE

Upon successful completion with a size not equal to 0, *realloc*( ) shall return a pointer to the (possibly moved) allocated space. If *size* is 0, either a null pointer or a unique pointer that can be successfully passed to *free*( ) shall be returned. If there is not enough available memory, *realloc*( ) CX shall return a null pointer and set *errno* to [ENOMEM].

ERRORS

The *realloc*( ) function shall fail if:

CX [ENOMEM] Insufficient memory is available.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

calloc( ), *free*( ), *malloc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

37126 **CHANGE HISTORY**

37127 First released in Issue 1. Derived from Issue 1 of the SVID.

37128 **Issue 6**

37129 Extensions beyond the ISO C standard are now marked.

37130 The following new requirements on POSIX implementations derive from alignment with the
37131 Single UNIX Specification:

37132 • In the RETURN VALUE section, if there is not enough available memory, the setting of *errno*
37133 to [ENOMEM] is added.

37134 • The [ENOMEM] error condition is added.

NAME

realpath — resolve a pathname

SYNOPSIS

XSI
```
#include <stdlib.h>

char *realpath(const char *restrict file_name,
    char *restrict resolved_name);
```

DESCRIPTION

The *realpath*( ) function shall derive, from the pathname pointed to by *file_name*, an absolute pathname that names the same file, whose resolution does not involve `'.'`, `'..'`, or symbolic links. The generated pathname shall be stored as a null-terminated string, up to a maximum of {PATH_MAX} bytes, in the buffer pointed to by *resolved_name*.

RETURN VALUE

Upon successful completion, *realpath*( ) shall return a pointer to the resolved name. Otherwise, *realpath*( ) shall return a null pointer and set *errno* to indicate the error, and the contents of the buffer pointed to by *resolved_name* are undefined.

ERRORS

The *realpath*( ) function shall fail if:

[EACCES] Read or search permission was denied for a component of *file_name*.

[EINVAL] Either the *file_name* or *resolved_name* argument is a null pointer.

[EIO] An error occurred while reading from the file system.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[ENAMETOOLONG]
The length of the *file_name* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

[ENOENT] A component of *file_name* does not name an existing file or *file_name* points to an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

The *realpath*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

[ENOMEM] Insufficient storage space is available.

EXAMPLES

Generating an Absolute Pathname

The following example generates an absolute pathname for the file identified by the *symlinkpath* argument. The generated pathname is stored in the *actualpath* array.

```
#include <stdlib.h>
...
char *symlinkpath = "/tmp/symlink/file";
char actualpath [PATH_MAX+1];
char *ptr;

ptr = realpath(symlinkpath, actualpath);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getcwd( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The **restrict** keyword is added to the *realpath*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

37198 **NAME**

37199 recv — receive a message from a connected socket

37200 **SYNOPSIS**

37201 `#include <sys/socket.h>`

37202 `ssize_t recv(int socket, void *buffer, size_t length, int flags);`

37203 **DESCRIPTION**

37204 The *recv*( ) function shall receive a message from a connection-mode or connectionless-mode
37205 socket. It is normally used with connected sockets because it does not permit the application to
37206 retrieve the source address of received data.

37207 The *recv*( ) function takes the following arguments:

37208 *socket* Specifies the socket file descriptor.

37209 *buffer* Points to a buffer where the message should be stored.

37210 *length* Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

37211 *flags* Specifies the type of message reception. Values of this argument are formed by
37212 logically OR'ing zero or more of the following values:

37213 MSG_PEEK Peeks at an incoming message. The data is treated as unread and
37214 the next *recv*( ) or similar function shall still return this data.

37215 MSG_OOB Requests out-of-band data. The significance and semantics of
37216 out-of-band data are protocol-specific.

37217 MSG_WAITALL On SOCK_STREAM sockets this requests that the function block |
37218 until the full amount of data can be returned. The function may |
37219 return the smaller amount of data if the socket is a message- |
37220 based socket, if a signal is caught, if the connection is |
37221 terminated, if MSG_PEEK was specified, or if an error is pending |
37222 for the socket. |

37223 The *recv*( ) function shall return the length of the message written to the buffer pointed to by the
37224 *buffer* argument. For message-based sockets, such as SOCK_DGRAM and SOCK_SEQPACKET,
37225 the entire message shall be read in a single operation. If a message is too long to fit in the
37226 supplied buffer, and MSG_PEEK is not set in the *flags* argument, the excess bytes shall be
37227 discarded. For stream-based sockets, such as SOCK_STREAM, message boundaries shall be
37228 ignored. In this case, data shall be returned to the user as soon as it becomes available, and no |
37229 data shall be discarded.

37230 If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first
37231 message.

37232 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file
37233 descriptor, *recv*( ) shall block until a message arrives. If no messages are available at the socket
37234 and O_NONBLOCK is set on the socket's file descriptor, *recv*( ) shall fail and set *errno* to
37235 [EAGAIN] or [EWOULDBLOCK].

37236 **RETURN VALUE**

37237 Upon successful completion, *recv*( ) shall return the length of the message in bytes. If no
37238 messages are available to be received and the peer has performed an orderly shutdown, *recv*( )
37239 shall return 0. Otherwise, –1 shall be returned and *errno* set to indicate the error.

37240 **ERRORS**

37241 The *recv*( ) function shall fail if:

37242 [EAGAIN] or [EWOULDBLOCK]
37243 The socket's file descriptor is marked O_NONBLOCK and no data is waiting
37244 to be received; or MSG_OOB is set and no out-of-band data is available and
37245 either the socket's file descriptor is marked O_NONBLOCK or the socket does
37246 not support blocking to await out-of-band data.

37247 [EBADF] The *socket* argument is not a valid file descriptor.

37248 [ECONNRESET] A connection was forcibly closed by a peer.

37249 [EINTR] The *recv*( ) function was interrupted by a signal that was caught, before any
37250 data was available.

37251 [EINVAL] The MSG_OOB flag is set and no out-of-band data is available.

37252 [ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.

37253 [ENOTSOCK] The *socket* argument does not refer to a socket.

37254 [EOPNOTSUPP] The specified flags are not supported for this socket type or protocol.

37255 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
37256 transmission timeout on active connection.

37257 The *recv*( ) function may fail if:

37258 [EIO] An I/O error occurred while reading from or writing to the file system.

37259 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

37260 [ENOMEM] Insufficient memory was available to fulfill the request.

37261 **EXAMPLES**

37262 None.

37263 **APPLICATION USAGE**

37264 The *recv*( ) function is equivalent to *recvfrom*( ) with a zero *address_len* argument, and to *read*( ) if |
37265 no flags are used.

37266 The *select*( ) and *poll*( ) functions can be used to determine when data is available to be received.

37267 **RATIONALE**

37268 None.

37269 **FUTURE DIRECTIONS**

37270 None.

37271 **SEE ALSO**

37272 *poll*( ), *read*( ), *recvmsg*( ), *recvfrom*( ), *select*( ), *send*( ), *sendmsg*( ), *sendto*( ), *shutdown*( ), *socket*( ),
37273 *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

37274 **CHANGE HISTORY**

37275 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

recvfrom — receive a message from a socket

SYNOPSIS

```
#include <sys/socket.h>

ssize_t recvfrom(int socket, void *restrict buffer, size_t length,
    int flags, struct sockaddr *restrict address,
    socklen_t *restrict address_len);
```

DESCRIPTION

The *recvfrom*( ) function shall receive a message from a connection-mode or connectionless-mode socket. It is normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data.

The *recvfrom*( ) function takes the following arguments:

socket Specifies the socket file descriptor.

buffer Points to the buffer where the message should be stored.

length Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

flags Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:

MSG_PEEK Peeks at an incoming message. The data is treated as unread and the next *recvfrom*( ) or similar function shall still return this data.

MSG_OOB Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.

MSG_WAITALL On SOCK_STREAM sockets this requests that the function block until the full amount of data can be returned. The function may return the smaller amount of data if the socket is a message-based socket, if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket.

address A null pointer, or points to a **sockaddr** structure in which the sending address is to be stored. The length and format of the address depend on the address family of the socket.

address_len Specifies the length of the **sockaddr** structure pointed to by the *address* argument.

The *recvfrom*( ) function shall return the length of the message written to the buffer pointed to by the *buffer* argument. For message-based sockets, such as SOCK_RAW, SOCK_DGRAM, and SOCK_SEQPACKET, the entire message shall be read in a single operation. If a message is too long to fit in the supplied buffer, and MSG_PEEK is not set in the *flags* argument, the excess bytes shall be discarded. For stream-based sockets, such as SOCK_STREAM, message boundaries shall be ignored. In this case, data shall be returned to the user as soon as it becomes available, and no data shall be discarded.

If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first message.

Not all protocols provide the source address for messages. If the *address* argument is not a null pointer and the protocol provides the source address of messages, the source address of the

received message shall be stored in the **sockaddr** structure pointed to by the *address* argument, and the length of this address shall be stored in the object pointed to by the *address_len* argument.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address shall be truncated.

If the *address* argument is not a null pointer and the protocol does not provide the source address of messages, the value stored in the object pointed to by *address* is unspecified.

If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, *recvfrom*( ) shall block until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, *recvfrom*( ) shall fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

RETURN VALUE

Upon successful completion, *recvfrom*( ) shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, *recvfrom*( ) shall return 0. Otherwise, the function shall return –1 and set *errno* to indicate the error.

ERRORS

The *recvfrom*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]
: The socket's file descriptor is marked O_NONBLOCK and no data is waiting to be received; or MSG_OOB is set and no out-of-band data is available and either the socket's file descriptor is marked O_NONBLOCK or the socket does not support blocking to await out-of-band data.

[EBADF]
: The *socket* argument is not a valid file descriptor.

[ECONNRESET]
: A connection was forcibly closed by a peer.

[EINTR]
: A signal interrupted *recvfrom*( ) before any data was available.

[EINVAL]
: The MSG_OOB flag is set and no out-of-band data is available.

[ENOTCONN]
: A receive is attempted on a connection-mode socket that is not connected.

[ENOTSOCK]
: The *socket* argument does not refer to a socket.

[EOPNOTSUPP]
: The specified flags are not supported for this socket type.

[ETIMEDOUT]
: The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The *recvfrom*( ) function may fail if:

[EIO]
: An I/O error occurred while reading from or writing to the file system.

[ENOBUFS]
: Insufficient resources were available in the system to perform the operation.

[ENOMEM]
: Insufficient memory was available to fulfill the request.

EXAMPLES

None.

APPLICATION USAGE

The *select*( ) and *poll*( ) functions can be used to determine when data is available to be received.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

poll( ), *read*( ), *recv*( ), *recvmsg*( ), *select*( ) (on page 1742)1 *send*( ), *sendmsg*( ), *sendto*( ), *shutdown*( ), *socket*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

37369 **NAME**

37370 recvmsg — receive a message from a socket

37371 **SYNOPSIS**

37372 `#include <sys/socket.h>`

37373 `ssize_t recvmsg(int socket, struct msghdr *message, int flags);`

37374 **DESCRIPTION**

37375 The *recvmsg*( ) function shall receive a message from a connection-mode or connectionless-mode
37376 socket. It is normally used with connectionless-mode sockets because it permits the application
37377 to retrieve the source address of received data.

37378 The *recvmsg*( ) function takes the following arguments:

37379 *socket* Specifies the socket file descriptor.

37380 *message* Points to a **msghdr** structure, containing both the buffer to store the source
37381 address and the buffers for the incoming message. The length and format of
37382 the address depend on the address family of the socket. The *msg_flags* member
37383 is ignored on input, but may contain meaningful values on output.

37384 *flags* Specifies the type of message reception. Values of this argument are formed
37385 by logically OR'ing zero or more of the following values:

37386 MSG_OOB Requests out-of-band data. The significance and semantics
37387 of out-of-band data are protocol-specific.

37388 MSG_PEEK Peeks at the incoming message.

37389 MSG_WAITALL On SOCK_STREAM sockets this requests that the function |
37390 block until the full amount of data can be returned. The |
37391 function may return the smaller amount of data if the socket |
37392 is a message-based socket, if a signal is caught, if the |
37393 connection is terminated, if MSG_PEEK was specified, or if |
37394 an error is pending for the socket. |

37395 The *recvmsg*( ) function shall receive messages from unconnected or connected sockets and shall |
37396 return the length of the message. |

37397 The *recvmsg*( ) function shall return the total length of the message. For message-based sockets,
37398 such as SOCK_DGRAM and SOCK_SEQPACKET, the entire message shall be read in a single
37399 operation. If a message is too long to fit in the supplied buffers, and MSG_PEEK is not set in the
37400 *flags* argument, the excess bytes shall be discarded, and MSG_TRUNC shall be set in the |
37401 *msg_flags* member of the **msghdr** structure. For stream-based sockets, such as SOCK_STREAM,
37402 message boundaries shall be ignored. In this case, data shall be returned to the user as soon as it |
37403 becomes available, and no data shall be discarded. |

37404 If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first
37405 message.

37406 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file
37407 descriptor, *recvmsg*( ) shall block until a message arrives. If no messages are available at the
37408 socket and O_NONBLOCK is set on the socket's file descriptor, *recvmsg*( ) function shall fail and
37409 set *errno* to [EAGAIN] or [EWOULDBLOCK].

37410 In the **msghdr** structure, the *msg_name* and *msg_namelen* members specify the source address if
37411 the socket is unconnected. If the socket is connected, the *msg_name* and *msg_namelen* members |
37412 shall be ignored. The *msg_name* member may be a null pointer if no names are desired or |
37413 required. The *msg_iov* and *msg_iovlen* fields are used to specify where the received data shall be

stored. *msg_iov* points to an array of **iovec** structures; *msg_iovlen* shall be set to the dimension of this array. In each **iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field gives its size in bytes. Each storage area indicated by *msg_iov* is filled with received data in turn until all of the received data is stored or all of the areas have been filled.

Upon successful completion, the *msg_flags* member of the message header shall be the bitwise-inclusive OR of all of the following flags that indicate conditions detected for the received message:

MSG_EOR	End of record was received (if supported by the protocol).
MSG_OOB	Out-of-band data was received.
MSG_TRUNC	Normal data was truncated.
MSG_CTRUNC	Control data was truncated.

RETURN VALUE

Upon successful completion, *recvmsg*( ) shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, *recvmsg*( ) shall return 0. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *recvmsg*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]	The socket's file descriptor is marked O_NONBLOCK and no data is waiting to be received; or MSG_OOB is set and no out-of-band data is available and either the socket's file descriptor is marked O_NONBLOCK or the socket does not support blocking to await out-of-band data.
[EBADF]	The *socket* argument is not a valid open file descriptor.
[ECONNRESET]	A connection was forcibly closed by a peer.
[EINTR]	This function was interrupted by a signal before any data was available.
[EINVAL]	The sum of the *iov_len* values overflows a **ssize_t**, or the MSG_OOB flag is set and no out-of-band data is available.
[EMSGSIZE]	The *msg_iovlen* member of the **msghdr** structure pointed to by *message* is less than or equal to 0, or is greater than {IOV_MAX}.
[ENOTCONN]	A receive is attempted on a connection-mode socket that is not connected.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The specified flags are not supported for this socket type.
[ETIMEDOUT]	The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The *recvmsg*( ) function may fail if:

[EIO]	An I/O error occurred while reading from or writing to the file system.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[ENOMEM]	Insufficient memory was available to fulfill the request.

37452 **EXAMPLES**

37453 None.

37454 **APPLICATION USAGE**

37455 The *select*( ) and *poll*( ) functions can be used to determine when data is available to be received.

37456 **RATIONALE**

37457 None.

37458 **FUTURE DIRECTIONS**

37459 None.

37460 **SEE ALSO**

37461 *poll*( ), *recv*( ), *recvfrom*( ), *select*( ), *send*( ), *sendmsg*( ), *sendto*( ), *shutdown*( ), *socket*( ), the Base
37462 Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

37463 **CHANGE HISTORY**

37464 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

regcomp, regerror, regexec, regfree — regular expression matching

SYNOPSIS

```
#include <regex.h>

int regcomp(regex_t *restrict preg, const char *restrict pattern, int cflags);
size_t regerror(int errcode, const regex_t *restrict preg,
    char *restrict errbuf, size_t errbuf_size);
int regexec(const regex_t *restrict preg, const char *restrict string,
    size_t nmatch, regmatch_t pmatch[restrict], int eflags);
void regfree(regex_t *preg);
```

DESCRIPTION

These functions interpret *basic* and *extended* regular expressions as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 9, Regular Expressions.

The **regex_t** structure is defined in **<regex.h>** and contains at least the following member:

Member Type	Member Name	Description
`size_t`	`re_nsub`	Number of parenthesized subexpressions.

The **regmatch_t** structure is defined in **<regex.h>** and contains at least the following members:

Member Type	Member Name	Description
regoff_t	*rm_so*	Byte offset from start of *string* to start of substring.
regoff_t	*rm_eo*	Byte offset from start of *string* of the first character after the end of substring.

The *regcomp*( ) function shall compile the regular expression contained in the string pointed to by the *pattern* argument and place the results in the structure pointed to by *preg*. The *cflags* argument is the bitwise-inclusive OR of zero or more of the following flags, which are defined in the **<regex.h>** header:

REG_EXTENDED — Use Extended Regular Expressions.

REG_ICASE — Ignore case in match. (See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 9, Regular Expressions.)

REG_NOSUB — Report only success/fail in *regexec*( ).

REG_NEWLINE — Change the handling of <newline>s, as described in the text.

The default regular expression type for *pattern* is a Basic Regular Expression. The application can specify Extended Regular Expressions using the REG_EXTENDED *cflags* flag.

If the REG_NOSUB flag was not set in *cflags*, then *regcomp*( ) shall set *re_nsub* to the number of parenthesized subexpressions (delimited by `"\(\)"` in basic regular expressions or `"( )"` in extended regular expressions) found in *pattern*.

The *regexec*( ) function compares the null-terminated string specified by *string* with the compiled regular expression *preg* initialized by a previous call to *regcomp*( ). If it finds a match, *regexec*( ) shall return 0; otherwise, it shall return non-zero indicating either no match or an error. The *eflags* argument is the bitwise-inclusive OR of zero or more of the following flags, which are defined in the **<regex.h>** header:

REG_NOTBOL The first character of the string pointed to by *string* is not the beginning of the line. Therefore, the circumflex character (`'^'`), when taken as a special character, shall not match the beginning of *string*.

REG_NOTEOL The last character of the string pointed to by *string* is not the end of the line. Therefore, the dollar sign (`'$'`), when taken as a special character, shall not match the end of *string*.

If *nmatch* is 0 or REG_NOSUB was set in the *cflags* argument to *regcomp*(), then *regexec*() shall ignore the *pmatch* argument. Otherwise, the application shall ensure that the *pmatch* argument points to an array with at least *nmatch* elements, and *regexec*() shall fill in the elements of that array with offsets of the substrings of *string* that correspond to the parenthesized subexpressions of *pattern*: *pmatch*[*i*].*rm_so* shall be the byte offset of the beginning and *pmatch*[*i*].*rm_eo* shall be one greater than the byte offset of the end of substring *i*. (Subexpression *i* begins at the *i*th matched open parenthesis, counting from 1.) Offsets in *pmatch*[0] identify the substring that corresponds to the entire regular expression. Unused elements of *pmatch* up to *pmatch*[*nmatch*−1] shall be filled with −1. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself counts as a subexpression), then *regexec*() shall still do the match, but shall record only the first *nmatch* substrings.

When matching a basic or extended regular expression, any given parenthesized subexpression of *pattern* might participate in the match of several different substrings of *string*, or it might not match any substring even though the pattern as a whole did match. The following rules shall be used to determine which substrings to report in *pmatch* when matching regular expressions:

1. If subexpression *i* in a regular expression is not contained within another subexpression, and it participated in the match several times, then the byte offsets in *pmatch*[*i*] shall delimit the last such match.

2. If subexpression *i* is not contained within another subexpression, and it did not participate in an otherwise successful match, the byte offsets in *pmatch*[*i*] shall be −1. A subexpression does not participate in the match when:

   `'*'` or `"\{\}"` appears immediately after the subexpression in a basic regular expression, or `'*'`, `'?'`, or `"{}"` appears immediately after the subexpression in an extended regular expression, and the subexpression did not match (matched 0 times)

   or:

   `'|'` is used in an extended regular expression to select this subexpression or another, and the other subexpression matched.

3. If subexpression *i* is contained within another subexpression *j*, and *i* is not contained within any other subexpression that is contained within *j*, and a match of subexpression *j* is reported in *pmatch*[*j*], then the match or non-match of subexpression *i* reported in *pmatch*[*i*] shall be as described in 1. and 2. above, but within the substring reported in *pmatch*[*j*] rather than the whole string. The offsets in *pmatch*[*i*] are still relative to the start of string.

4. If subexpression *i* is contained in subexpression *j*, and the byte offsets in *pmatch*[*j*] are −1, then the pointers in *pmatch*[*i*] shall also be −1.

5. If subexpression *i* matched a zero-length string, then both byte offsets in *pmatch*[*i*] shall be the byte offset of the character or null terminator immediately following the zero-length string.

If, when *regexec*() is called, the locale is different from when the regular expression was compiled, the result is undefined.

If REG_NEWLINE is not set in *cflags*, then a <newline> in *pattern* or *string* shall be treated as an ordinary character. If REG_NEWLINE is set, then <newline> shall be treated as an ordinary character except as follows:

1. A <newline> in *string* shall not be matched by a period outside a bracket expression or by any form of a non-matching list (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 9, Regular Expressions).
2. A circumflex (`'^'`) in *pattern*, when used to specify expression anchoring (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3.8, BRE Expression Anchoring), shall match the zero-length string immediately after a <newline> in *string*, regardless of the setting of REG_NOTBOL.
3. A dollar sign (`'$'`) in *pattern*, when used to specify expression anchoring, shall match the zero-length string immediately before a <newline> in *string*, regardless of the setting of REG_NOTEOL.

The *regfree*( ) function frees any memory allocated by *regcomp*( ) associated with *preg*.

The following constants are defined as error return values:

REG_NOMATCH	*regexec*( ) failed to match.
REG_BADPAT	Invalid regular expression.
REG_ECOLLATE	Invalid collating element referenced.
REG_ECTYPE	Invalid character class type referenced.
REG_EESCAPE	Trailing `'\'` in pattern.
REG_ESUBREG	Number in `"\digit"` invalid or in error.
REG_EBRACK	`"[]"` imbalance.
REG_EPAREN	`"\(\)"` or `"()"` imbalance.
REG_EBRACE	`"\{\}"` imbalance.
REG_BADBR	Content of `"\{\}"` invalid: not a number, number too large, more than two numbers, first larger than second.
REG_ERANGE	Invalid endpoint in range expression.
REG_ESPACE	Out of memory.
REG_BADRPT	`'?'`, `'*'`, or `'+'` not preceded by valid regular expression.

The *regerror*( ) function provides a mapping from error codes returned by *regcomp*( ) and *regexec*( ) to unspecified printable strings. It generates a string corresponding to the value of the *errcode* argument, which the application shall ensure is the last non-zero value returned by *regcomp*( ) or *regexec*( ) with the given value of *preg*. If *errcode* is not such a value, the content of the generated string is unspecified.

If *preg* is a null pointer, but *errcode* is a value returned by a previous call to *regexec*( ) or *regcomp*( ), the *regerror*( ) still generates an error string corresponding to the value of *errcode*, but it might not be as detailed under some implementations.

If the *errbuf_size* argument is not 0, *regerror*( ) shall place the generated string into the buffer of size *errbuf_size* bytes pointed to by *errbuf*. If the string (including the terminating null) cannot fit in the buffer, *regerror*( ) shall truncate the string and null-terminates the result.

If *errbuf_size* is 0, *regerror*( ) shall ignore the *errbuf* argument, and return the size of the buffer needed to hold the generated string.

If the *preg* argument to *regexec*( ) or *regfree*( ) is not a compiled regular expression returned by *regcomp*( ), the result is undefined. A *preg* is no longer treated as a compiled regular expression after it is given to *regfree*( ).

RETURN VALUE

Upon successful completion, the *regcomp*( ) function shall return 0. Otherwise, it shall return an integer value indicating an error as described in **<regex.h>**, and the content of *preg* is undefined. If a code is returned, the interpretation shall be as given in **<regex.h>**.

If *regcomp*( ) detects an invalid RE, it may return REG_BADPAT, or it may return one of the error codes that more precisely describes the error.

Upon successful completion, the *regexec*( ) function shall return 0. Otherwise, it shall return REG_NOMATCH to indicate no match.

Upon successful completion, the *regerror*( ) function shall return the number of bytes needed to hold the entire generated string, including the null termination. If the return value is greater than *errbuf_size*, the string returned in the buffer pointed to by *errbuf* has been truncated.

The *regfree*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

```
#include <regex.h>

/*
 * Match string against the extended regular expression in
 * pattern, treating errors as no match.
 *
 * Return 1 for match, 0 for no match.
 */

int
match(const char *string, char *pattern)
{
    int    status;
    regex_t    re;

    if (regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB) != 0) {
        return(0);      /* Report error. */
    }
    status = regexec(&re, string, (size_t) 0, NULL, 0);
    regfree(&re);
    if (status != 0) {
        return(0);      /* Report error. */
    }
    return(1);
}
```

The following demonstrates how the REG_NOTBOL flag could be used with *regexec*( ) to find all substrings in a line that match a pattern supplied by a user. (For simplicity of the example, very little error checking is done.)

```
(void) regcomp (&re, pattern, 0);
/* This call to regexec() finds the first match on the line. */
error = regexec (&re, &buffer[0], 1, &pm, 0);
while (error == 0) {  /* While matches found. */
    /* Substring found between pm.rm_so and pm.rm_eo. */
    /* This call to regexec() finds the next match. */
    error = regexec (&re, buffer + pm.rm_eo, 1, &pm, REG_NOTBOL);
}
```

APPLICATION USAGE

An application could use:

```
regerror(code,preg,(char *)NULL,(size_t)0)
```

to find out how big a buffer is needed for the generated string, *malloc*() a buffer to hold the string, and then call *regerror*() again to get the string. Alternatively, it could allocate a fixed, static buffer that is big enough to hold most strings, and then use *malloc*() to allocate a larger buffer if it finds that this is too small.

To match a pattern as described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13, Pattern Matching Notation, use the *fnmatch*() function.

RATIONALE

The *regmatch*() function must fill in all *nmatch* elements of *pmatch*, where *nmatch* and *pmatch* are supplied by the application, even if some elements of *pmatch* do not correspond to subexpressions in *pattern*. The application writer should note that there is probably no reason for using a value of *nmatch* that is larger than *preg*–>*re_nsub*+1.

The REG_NEWLINE flag supports a use of RE matching that is needed in some applications like text editors. In such applications, the user supplies an RE asking the application to find a line that matches the given expression. An anchor in such an RE anchors at the beginning or end of any line. Such an application can pass a sequence of <newline>-separated lines to *regexec*() as a single long string and specify REG_NEWLINE to *regcomp*() to get the desired behavior. The application must ensure that there are no explicit <newline>s in *pattern* if it wants to ensure that any match occurs entirely within a single line.

The REG_NEWLINE flag affects the behavior of *regexec*(), but it is in the *cflags* parameter to *regcomp*() to allow flexibility of implementation. Some implementations will want to generate the same compiled RE in *regcomp*() regardless of the setting of REG_NEWLINE and have *regexec*() handle anchors differently based on the setting of the flag. Other implementations will generate different compiled REs based on the REG_NEWLINE.

The REG_ICASE flag supports the operations taken by the *grep* **–i** option and the historical implementations of *ex* and *vi*. Including this flag will make it easier for application code to be written that does the same thing as these utilities.

The substrings reported in *pmatch*[] are defined using offsets from the start of the string rather than pointers. Since this is a new interface, there should be no impact on historical implementations or applications, and offsets should be just as easy to use as pointers. The change to offsets was made to facilitate future extensions in which the string to be searched is presented to *regexec*() in blocks, allowing a string to be searched that is not all in memory at once.

A new type **regoff_t** is used for the elements of *pmatch*[] to ensure that the application can represent either the largest possible array in memory (important for an application conforming to the Shell and Utilities volume of IEEE Std 1003.1-200x) or the largest possible file (important for an application using the extension where a file is searched in chunks).

The standard developers rejected the inclusion of a *regsub*( ) function that would be used to do substitutions for a matched RE. While such a routine would be useful to some applications, its utility would be much more limited than the matching function described here. Both RE parsing and substitution are possible to implement without support other than that required by the ISO C standard, but matching is much more complex than substituting. The only difficult part of substitution, given the information supplied by *regexec*( ), is finding the next character in a string when there can be multi-byte characters. That is a much larger issue, and one that needs a more general solution.

The *errno* variable has not been used for error returns to avoid filling the *errno* name space for this feature.

The interface is defined so that the matched substrings *rm_sp* and *rm_ep* are in a separate **regmatch_t** structure instead of in **regex_t**. This allows a single compiled RE to be used simultaneously in several contexts; in *main*( ) and a signal handler, perhaps, or in multiple threads of lightweight processes. (The *preg* argument to *regexec*( ) is declared with type **const**, so the implementation is not permitted to use the structure to store intermediate results.) It also allows an application to request an arbitrary number of substrings from an RE. The number of subexpressions in the RE is reported in *re_nsub* in *preg*. With this change to *regexec*( ), consideration was given to dropping the REG_NOSUB flag since the user can now specify this with a zero *nmatch* argument to *regexec*( ). However, keeping REG_NOSUB allows an implementation to use a different (perhaps more efficient) algorithm if it knows in *regcomp*( ) that no subexpressions need be reported. The implementation is only required to fill in *pmatch* if *nmatch* is not zero and if REG_NOSUB is not specified. Note that the **size_t** type, as defined in the ISO C standard, is unsigned, so the description of *regexec*( ) does not need to address negative values of *nmatch*.

REG_NOTBOL was added to allow an application to do repeated searches for the same pattern in a line. If the pattern contains a circumflex character that should match the beginning of a line, then the pattern should only match when matched against the beginning of the line. Without the REG_NOTBOL flag, the application could rewrite the expression for subsequent matches, but in the general case this would require parsing the expression. The need for REG_NOTEOL is not as clear; it was added for symmetry.

The addition of the *regerror*( ) function addresses the historical need for conforming application programs to have access to error information more than ''Function failed to compile/match your RE for unknown reasons''.

This interface provides for two different methods of dealing with error conditions. The specific error codes (REG_EBRACE, for example), defined in **<regex.h>**, allow an application to recover from an error if it is so able. Many applications, especially those that use patterns supplied by a user, will not try to deal with specific error cases, but will just use *regerror*( ) to obtain a human-readable error message to present to the user.

The *regerror*( ) function uses a scheme similar to *confstr*( ) to deal with the problem of allocating memory to hold the generated string. The scheme used by *strerror*( ) in the ISO C standard was considered unacceptable since it creates difficulties for multi-threaded applications.

The *preg* argument is provided to *regerror*( ) to allow an implementation to generate a more descriptive message than would be possible with *errcode* alone. An implementation might, for example, save the character offset of the offending character of the pattern in a field of *preg*, and then include that in the generated message string. The implementation may also ignore *preg*.

A REG_FILENAME flag was considered, but omitted. This flag caused *regexec*( ) to match patterns as described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13, Pattern Matching Notation instead of REs. This service is now provided by the *fnmatch*( )

function.

Notice that there is a difference in philosophy between the ISO POSIX-2:1993 standard and IEEE Std 1003.1-200x in how to handle a bad regular expression. The ISO POSIX-2:1993 standard says that many bad constructs produce undefined results, or that the interpretation is undefined. IEEE Std 1003.1-200x, however, says that the interpretation of such REs is unspecified. The term ''undefined'' means that the action by the application is an error, of similar severity to passing a bad pointer to a function.

The *regcomp*( ) and *regexec*( ) functions are required to accept any null-terminated string as the *pattern* argument. If the meaning of the string is undefined, the behavior of the function is unspecified. IEEE Std 1003.1-200x does not specify how the functions will interpret the pattern; they might return error codes, or they might do pattern matching in some completely unexpected way, but they should not do something like abort the process.

FUTURE DIRECTIONS

None.

SEE ALSO

fnmatch( ), *glob*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<regex.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

Moved from POSIX2 C-language Binding to BASE.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

The REG_ENOSYS constant is removed.

The **restrict** keyword is added to the *regcomp*( ), *regerror*( ), and *regexec*( ) prototypes for alignment with the ISO/IEC 9899:1999 standard.

37764 **NAME**

37765 remainder, remainderf, remainderl — remainder function

37766 **SYNOPSIS**

37767 `#include <math.h>`

37768 `double remainder(double x, double y);`
37769 `float remainderf(float x, float y);`
37770 `long double remainderl(long double x, long double y);`

37771 **DESCRIPTION**

37772 CX The functionality described on this reference page is aligned with the ISO C standard. Any
37773 conflict between the requirements described here and the ISO C standard is unintentional. This
37774 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

37775 These functions shall return the floating-point remainder $r=x-ny$ when y is non-zero. The value
37776 n is the integral value nearest the exact value x/y. When $|n-x/y|=\frac{1}{2}$, the value n is chosen to
37777 be even.

37778 The behavior of *remainder*( ) shall be independent of the rounding mode.

37779 **RETURN VALUE**

37780 Upon successful completion, these functions shall return the floating-point remainder $r=x-ny$
37781 when y is non-zero.

37782 MX If x or y is NaN, a NaN shall be returned.

37783 If x is infinite or y is 0 and the other is non-NaN, a domain error shall occur, and either a NaN (if
37784 supported), or an implementation-defined value shall be returned.

37785 **ERRORS**

37786 These functions shall fail if:

37787 MX Domain Error — The x argument is ±Inf, or the y argument is ±0 and the other argument is
37788 non-NaN.

37789 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
37790 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling |
37791 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception |
37792 shall be raised. |

37793 **EXAMPLES**

37794 None.

37795 **APPLICATION USAGE**

37796 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
37797 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

37798 **RATIONALE**

37799 None.

37800 **FUTURE DIRECTIONS**

37801 None.

37802 **SEE ALSO**

37803 *abs*( ), *div*( ), *feclearexcept*( ), *fetestexcept*( ), *ldiv*( ), the Base Definitions volume of |
37804 IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, |
37805 **<math.h>**

37806 **CHANGE HISTORY**

37807 First released in Issue 4, Version 2.

37808 **Issue 5**

37809 Moved from X/OPEN UNIX extension to BASE.

37810 **Issue 6**

37811 The *remainder*( ) function is no longer marked as an extension.

37812 The *remainderf*( ) and *remainderl*( ) functions are added for alignment with the ISO/IEC 9899:1999
37813 standard.

37814 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
37815 revised to align with the ISO/IEC 9899:1999 standard.

37816 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
37817 marked.

NAME

remove — remove a file

SYNOPSIS

```
#include <stdio.h>

int remove(const char *path);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *remove*( ) function shall cause the file named by the pathname pointed to by path to be no longer accessible by that name. A subsequent attempt to open that file using that name shall fail, unless it is created anew.

CX If *path* does not name a directory, *remove*(*path*) shall be equivalent to *unlink*(*path*).

If *path* names a directory, *remove*(*path*) shall be equivalent to *rmdir*(*path*).

RETURN VALUE

CX Refer to *rmdir*( ) or *unlink*( ).

ERRORS

CX Refer to *rmdir*( ) or *unlink*( ).

EXAMPLES

Removing Access to a File

The following example shows how to remove access to a file named **/home/cnd/old_mods**.

```
#include <stdio.h>

int status;
...
status = remove("/home/cnd/old_mods");
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

rmdir( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard and the ISO C standard.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION, RETURN VALUE, and ERRORS sections are updated so that if *path* is not a directory, *remove*() is equivalent to *unlink*(), and if it is a directory, it is equivalent to *rmdir*().

37861 **NAME**

37862 remque — remove an element from a queue

37863 **SYNOPSIS**

37864 XSI
```
#include <search.h>
```

37865
```
void remque(void *element);
```

37866

37867 **DESCRIPTION**

37868 Refer to *insque*( ).

37869 **NAME**

37870 remquo, remquof, remquol — remainder functions

37871 **SYNOPSIS**

37872
```
#include <math.h>

double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y, int *quo);
```

37876 **DESCRIPTION**

37877 CX The functionality described on this reference page is aligned with the ISO C standard. Any
37878 conflict between the requirements described here and the ISO C standard is unintentional. This
37879 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

37880 The *remquo*( ), *remquof*( ), and *remquol*( ) functions shall compute the same remainder as the
37881 *remainder*( ), *remainderf*( ), and *remainderl*( ) functions, respectively. In the object pointed to by
37882 *quo*, they store a value whose sign is the sign of x/y and whose magnitude is congruent modulo
37883 2^n to the magnitude of the integral quotient of x/y, where n is an implementation-defined
37884 integer greater than or equal to 3.

37885 An application wishing to check for error situations should set *errno* to zero and call
37886 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
37887 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
37888 zero, an error has occurred.

37889 **RETURN VALUE**

37890 These functions shall return *x* REM *y*.

37891 MX If *x* or *y* is NaN, a NaN shall be returned.

37892 If *x* is ±Inf or *y* is zero and the other argument is non-NaN, a domain error shall occur, and either
37893 a NaN (if supported), or an implementation-defined value shall be returned.

37894 **ERRORS**

37895 These functions shall fail if:

37896 MX Domain Error — The *x* argument is ±Inf, or the *y* argument is ±0 and the other argument is
37897 non-NaN.

37898 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
37899 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling |
37900 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception |
37901 shall be raised. |

37902 **EXAMPLES**

37903 None.

37904 **APPLICATION USAGE**

37905 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
37906 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

37907 **RATIONALE**

37908 These functions are intended for implementing argument reductions which can exploit a few
37909 low-order bits of the quotient. Note that *x* may be so large in magnitude relative to *y* that an
37910 exact representation of the quotient is not practical.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *remainder*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

rename — rename a file

SYNOPSIS

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *rename*( ) function shall change the name of a file. The *old* argument points to the pathname of the file to be renamed. The *new* argument points to the new pathname of the file.

CX If either the *old* or *new* argument names a symbolic link, *rename*( ) shall operate on the symbolic link itself, and shall not resolve the last component of the argument. If the *old* argument and the *new* argument resolve to the same existing file, *rename*( ) shall return successfully and perform no other action.

If the *old* argument points to the pathname of a file that is not a directory, the *new* argument shall not point to the pathname of a directory. If the link named by the *new* argument exists, it shall be removed and *old* renamed to *new*. In this case, a link named *new* shall remain visible to other processes throughout the renaming operation and refer either to the file referred to by *new* or *old* before the operation began. Write access permission is required for both the directory containing *old* and the directory containing *new*.

If the *old* argument points to the pathname of a directory, the *new* argument shall not point to the pathname of a file that is not a directory. If the directory named by the *new* argument exists, it shall be removed and *old* renamed to *new*. In this case, a link named *new* shall exist throughout the renaming operation and shall refer either to the directory referred to by *new* or *old* before the operation began. If *new* names an existing directory, it shall be required to be an empty directory.

If the *old* argument points to a pathname of a symbolic link, the symbolic link shall be renamed. If the *new* argument points to a pathname of a symbolic link, the symbolic link shall be removed.

The *new* pathname shall not contain a path prefix that names *old*. Write access permission is required for the directory containing *old* and the directory containing *new*. If the *old* argument points to the pathname of a directory, write access permission may be required for the directory named by *old*, and, if it exists, the directory named by *new*.

If the link named by the *new* argument exists and the file's link count becomes 0 when it is removed and no process has the file open, the space occupied by the file shall be freed and the file shall no longer be accessible. If one or more processes have the file open when the last link is removed, the link shall be removed before *rename*( ) returns, but the removal of the file contents shall be postponed until all references to the file are closed.

Upon successful completion, *rename*( ) shall mark for update the *st_ctime* and *st_mtime* fields of the parent directory of each file.

If the *rename*( ) function fails for any reason other than [EIO], any file named by *new* shall be unaffected.

RETURN VALUE

CX Upon successful completion, *rename*( ) shall return 0; otherwise, –1 shall be returned, *errno* shall be set to indicate the error, and neither the file named by *old* nor the file named by *new* shall be changed or created.

37963 **ERRORS**

37964 The *rename*( ) function shall fail if:

37965 CX [EACCES] A component of either path prefix denies search permission; or one of the
37966 directories containing *old* or *new* denies write permissions; or, write
37967 permission is required and is denied for a directory pointed to by the *old* or
37968 *new* arguments.

37969 CX [EBUSY] The directory named by *old* or *new* is currently in use by the system or another
37970 process, and the implementation considers this an error.

37971 CX [EEXIST] or [ENOTEMPTY]
37972 The link named by *new* is a directory that is not an empty directory.

37973 CX [EINVAL] The *new* directory pathname contains a path prefix that names the *old* |
37974 directory.

37975 CX [EIO] A physical I/O error has occurred.

37976 CX [EISDIR] The *new* argument points to a directory and the *old* argument points to a file
37977 that is not a directory.

37978 CX [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
37979 argument.

37980 CX [EMLINK] The file named by *old* is a directory, and the link count of the parent directory
37981 of *new* would exceed {LINK_MAX}.

37982 CX [ENAMETOOLONG]
37983 The length of the *old* or *new* argument exceeds {PATH_MAX} or a pathname |
37984 component is longer than {NAME_MAX}. |

37985 CX [ENOENT] The link named by *old* does not name an existing file, or either *old* or *new*
37986 points to an empty string.

37987 CX [ENOSPC] The directory that would contain *new* cannot be extended.

37988 CX [ENOTDIR] A component of either path prefix is not a directory; or the *old* argument
37989 names a directory and *new* argument names a non-directory file.

37990 XSI [EPERM] or [EACCES]
37991 The S_ISVTX flag is set on the directory containing the file referred to by *old*
37992 and the caller is not the file owner, nor is the caller the directory owner, nor
37993 does the caller have appropriate privileges; or *new* refers to an existing file, the
37994 S_ISVTX flag is set on the directory containing this file, and the caller is not
37995 the file owner, nor is the caller the directory owner, nor does the caller have
37996 appropriate privileges.

37997 CX [EROFS] The requested operation requires writing in a directory on a read-only file
37998 system.

37999 CX [EXDEV] The links named by *new* and *old* are on different file systems and the
38000 implementation does not support links between file systems.

38001 The *rename*( ) function may fail if:

38002 XSI [EBUSY] The file named by the *old* or *new* arguments is a named STREAM.

38003 CX [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
38004 resolution of the *path* argument.

CX [ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

CX [ETXTBSY] The file to be renamed is a pure procedure (shared text) file that is being executed.

EXAMPLES

Renaming a File

The following example shows how to rename a file named **/home/cnd/mod1** to **/home/cnd/mod2**.

```
#include <stdio.h>

int status;
...
status = rename("/home/cnd/mod1", "/home/cnd/mod2");
```

APPLICATION USAGE

Some implementations mark for update the *st_ctime* field of renamed files and some do not. Applications which make use of the *st_ctime* field may behave differently with respect to renamed files unless they are designed to allow for either behavior.

RATIONALE

This *rename*( ) function is equivalent for regular files to that defined by the ISO C standard. Its inclusion here expands that definition to include actions on directories and specifies behavior when the *new* parameter names a file that already exists. That specification requires that the action of the function be atomic.

One of the reasons for introducing this function was to have a means of renaming directories while permitting implementations to prohibit the use of *link*( ) and *unlink*( ) with directories, thus constraining links to directories to those made by *mkdir*( ).

The specification that if *old* and *new* refer to the same file is intended to guarantee that:

```
rename("x", "x");
```

does not remove the file.

Renaming dot or dot-dot is prohibited in order to prevent cyclical file system paths.

See also the descriptions of [ENOTEMPTY] and [ENAMETOOLONG] in *rmdir*( ) and [EBUSY] in *unlink*( ). For a discussion of [EXDEV], see *link*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

link( ), *rmdir*( ), *symlink*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The [EBUSY] error is added to the ‘‘may fail’’ part of the ERRORS section.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The [EIO] mandatory error condition is added.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.
- The [ETXTBSY] optional error condition is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Details are added regarding the treatment of symbolic links.
- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

38058 **NAME**

38059 rewind — reset file position indicator in a stream

38060 **SYNOPSIS**

38061 `#include <stdio.h>`

38062 `void rewind(FILE *stream);`

38063 **DESCRIPTION**

38064 CX The functionality described on this reference page is aligned with the ISO C standard. Any
38065 conflict between the requirements described here and the ISO C standard is unintentional. This
38066 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

38067 The call:

38068 `rewind(stream)`

38069 shall be equivalent to:

38070 `(void) fseek(stream, 0L, SEEK_SET)`

38071 except that *rewind*( ) shall also clear the error indicator. |

38072 CX Since *rewind*( ) does not return a value, an application wishing to detect errors should clear *errno*, |
38073 then call *rewind*( ), and if *errno* is non-zero, assume an error has occurred.

38074 **RETURN VALUE**

38075 The *rewind*( ) function shall not return a value.

38076 **ERRORS**

38077 CX Refer to *fseek*( ) with the exception of [EINVAL] which does not apply.

38078 **EXAMPLES**

38079 None.

38080 **APPLICATION USAGE**

38081 None.

38082 **RATIONALE**

38083 None.

38084 **FUTURE DIRECTIONS**

38085 None.

38086 **SEE ALSO**

38087 *fseek*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

38088 **CHANGE HISTORY**

38089 First released in Issue 1. Derived from Issue 1 of the SVID.

38090 **Issue 6**

38091 Extensions beyond the ISO C standard are now marked.

38092 **NAME**

38093 rewinddir — reset position of directory stream to the beginning of a directory

38094 **SYNOPSIS**

38095
```
#include <dirent.h>
```

38096
```
void rewinddir(DIR *dirp);
```

38097 **DESCRIPTION**

38098 The *rewinddir*( ) function shall reset the position of the directory stream to which *dirp* refers to
38099 the beginning of the directory. It shall also cause the directory stream to refer to the current state
38100 of the corresponding directory, as a call to *opendir*( ) would have done. If *dirp* does not refer to a
38101 directory stream, the effect is undefined.

38102 After a call to the *fork*( ) function, either the parent or child (but not both) may continue
38103 XSI processing the directory stream using *readdir*( ), *rewinddir*( ), or *seekdir*( ). If both the parent and
38104 child processes use these functions, the result is undefined.

38105 **RETURN VALUE**

38106 The *rewinddir*( ) function shall not return a value.

38107 **ERRORS**

38108 No errors are defined.

38109 **EXAMPLES**

38110 None.

38111 **APPLICATION USAGE**

38112 The *rewinddir*( ) function should be used in conjunction with *opendir*( ), *readdir*( ), and *closedir*( ) to
38113 examine the contents of the directory. This method is recommended for portability.

38114 **RATIONALE**

38115 None.

38116 **FUTURE DIRECTIONS**

38117 None.

38118 **SEE ALSO**

38119 *closedir*( ), *opendir*( ), *readdir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**
38120 **<sys/types.h>**

38121 **CHANGE HISTORY**

38122 First released in Issue 2.

38123 **Issue 6**

38124 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

38125 The following new requirements on POSIX implementations derive from alignment with the
38126 Single UNIX Specification:

38127 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
38128 required for conforming implementations of previous POSIX specifications, it was not
38129 required for UNIX applications.

38130 **NAME**

38131 rindex — character string operations (**LEGACY**)

38132 **SYNOPSIS**

38133 XSI
```
#include <strings.h>
```

38134
```
char *rindex(const char *s, int c);
```

38135

38136 **DESCRIPTION**

38137 The *rindex*( ) function shall be equivalent to *strrchr*( ).

38138 **RETURN VALUE**

38139 Refer to *strrchr*( ).

38140 **ERRORS**

38141 Refer to *strrchr*( ).

38142 **EXAMPLES**

38143 None.

38144 **APPLICATION USAGE**

38145 *strrchr*( ) is preferred over this function.

38146 For maximum portability, it is recommended to replace the function call to *rindex*( ) as follows:

38147
```
#define rindex(a,b) strrchr((a),(b))
```

38148 **RATIONALE**

38149 None.

38150 **FUTURE DIRECTIONS**

38151 This function may be withdrawn in a future version.

38152 **SEE ALSO**

38153 *strrchr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

38154 **CHANGE HISTORY**

38155 First released in Issue 4, Version 2.

38156 **Issue 5**

38157 Moved from X/OPEN UNIX extension to BASE.

38158 **Issue 6**

38159 This function is marked LEGACY.

38160 **NAME**

38161 rint, rintf, rintl — round-to-nearest integral value

38162 **SYNOPSIS**

38163 `#include <math.h>`

38164 `double rint(double x);`
38165 `float rintf(float x);`
38166 `long double rintl(long double x);`

38167 **DESCRIPTION**

38168 CX The functionality described on this reference page is aligned with the ISO C standard. Any
38169 conflict between the requirements described here and the ISO C standard is unintentional. This
38170 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

38171 These functions shall return the integral value (represented as a **double**) nearest *x* in the
38172 direction of the current rounding mode. The current rounding mode is implementation-defined.

38173 If the current rounding mode rounds toward negative infinity, then *rint*( ) shall be equivalent to |
38174 *floor*( ). If the current rounding mode rounds toward positive infinity, then *rint*( ) shall be |
38175 equivalent to *ceil*( ). |

38176 These functions differ from the *nearbyint*( ), *nearbyintf*( ), and *nearbyintl*( ) functions only in that
38177 they may raise the inexact floating-point exception if the result differs in value from the
38178 argument.

38179 An application wishing to check for error situations should set *errno* to zero and call
38180 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
38181 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
38182 zero, an error has occurred.

38183 **RETURN VALUE**

38184 Upon successful completion, these functions shall return the integer (represented as a double
38185 precision number) nearest *x* in the direction of the current rounding mode.

38186 MX If *x* is NaN, a NaN shall be returned.

38187 If *x* is ±0, or ±Inf, *x* shall be returned.

38188 XSI If the correct value would cause overflow, a range error shall occur and *rint*( ), *rintf*( ), and *rintl*( )
38189 shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

38190 **ERRORS**

38191 These functions shall fail if:

38192 XSI

Range Error	The result would cause an overflow.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

abs( ), *ceil*( ), *feclearexcept*( ), *fetestexcept*( ), *nearbyint*( ), *floor*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- The *rintf*( ) and *rintl*( ) functions are added.
- The *rint*( ) function is no longer marked as an extension.
- The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899: 1999 standard.

  IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are marked.

38222 **NAME**

38223 rmdir — remove a directory

38224 **SYNOPSIS**

38225 `#include <unistd.h>`

38226 `int rmdir(const char *path);`

38227 **DESCRIPTION**

38228 The *rmdir*( ) function shall remove a directory whose name is given by *path*. The directory shall
38229 be removed only if it is an empty directory.

38230 If the directory is the root directory or the current working directory of any process, it is
38231 unspecified whether the function succeeds, or whether it shall fail and set *errno* to [EBUSY].

38232 If *path* names a symbolic link, then *rmdir*( ) shall fail and set *errno* to [ENOTDIR].

38233 If the *path* argument refers to a path whose final component is either dot or dot-dot, *rmdir*( ) shall
38234 fail.

38235 If the directory's link count becomes 0 and no process has the directory open, the space occupied
38236 by the directory shall be freed and the directory shall no longer be accessible. If one or more
38237 processes have the directory open when the last link is removed, the dot and dot-dot entries, if
38238 present, shall be removed before *rmdir*( ) returns and no new entries may be created in the
38239 directory, but the directory shall not be removed until all references to the directory are closed.

38240 If the directory is not an empty directory, *rmdir*( ) shall fail and set *errno* to [EEXIST] or
38241 [ENOTEMPTY].

38242 Upon successful completion, the *rmdir*( ) function shall mark for update the *st_ctime* and
38243 *st_mtime* fields of the parent directory.

38244 **RETURN VALUE**

38245 Upon successful completion, the function *rmdir*( ) shall return 0. Otherwise, –1 shall be returned,
38246 and *errno* set to indicate the error. If –1 is returned, the named directory shall not be changed.

38247 **ERRORS**

38248 The *rmdir*( ) function shall fail if:

38249 [EACCES] Search permission is denied on a component of the path prefix, or write
38250 permission is denied on the parent directory of the directory to be removed.

38251 [EBUSY] The directory to be removed is currently in use by the system or some process
38252 and the implementation considers this to be an error.

38253 [EEXIST] or [ENOTEMPTY]
38254 The *path* argument names a directory that is not an empty directory, or there
38255 are hard links to the directory other than dot or a single entry in dot-dot.

38256 [EINVAL] The *path* argument contains a last component that is dot.

38257 [EIO] A physical I/O error has occurred.

38258 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
38259 argument.

38260 [ENAMETOOLONG]
38261 The length of the *path* argument exceeds {PATH_MAX} or a pathname
38262 component is longer than

38263 NAME_MAX

[ENOENT] A component of *path* does not name an existing file, or the *path* argument names a nonexistent directory or points to an empty string.

[ENOTDIR] A component of *path* is not a directory.

XSI [EPERM] or [EACCES]
The S_ISVTX flag is set on the parent directory of the directory to be removed and the caller is not the owner of the directory to be removed, nor is the caller the owner of the parent directory, nor does the caller have the appropriate privileges.

[EROFS] The directory entry to be removed resides on a read-only file system.

The *rmdir*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}. |

EXAMPLES

Removing a Directory

The following example shows how to remove a directory named **/home/cnd/mod1**.

```
#include <unistd.h>

int status;
...
status = rmdir("/home/cnd/mod1");
```

APPLICATION USAGE

None.

RATIONALE

The *rmdir*( ) and *rename*( ) functions originated in 4.2 BSD, and they used [ENOTEMPTY] for the condition when the directory to be removed does not exist or *new* already exists. When the 1984 /usr/group standard was published, it contained [EEXIST] instead. When these functions were adopted into System V, the 1984 /usr/group standard was used as a reference. Therefore, several existing applications and implementations support/use both forms, and no agreement could be reached on either value. All implementations are required to supply both [EEXIST] and [ENOTEMPTY] in **<errno.h>** with distinct values, so that applications can use both values in C-language **case** statements.

The meaning of deleting *pathname*/**dot** is unclear, because the name of the file (directory) in the parent directory to be removed is not clear, particularly in the presence of multiple links to a directory.

IEEE Std 1003.1-200x was silent with regard to the behavior of *rmdir*( ) when there are multiple hard links to the directory being removed. The requirement to set *errno* to [EEXIST] or [ENOTEMPTY] clarifies the behavior in this case.

If the process' current working directory is being removed, that should be an allowed error. |

Virtually all existing implementations detect [ENOTEMPTY] or the case of dot-dot. The text in Section 2.3 (on page 471) about returning any one of the possible errors permits that behavior to continue. The [ELOOP] error may be returned if more than {SYMLOOP_MAX} symbolic links

are encountered during resolution of the *path* argument.

FUTURE DIRECTIONS

None.

SEE ALSO

mkdir( ), *remove*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The DESCRIPTION is updated to indicate the results of naming a symbolic link in *path*.
- The [EIO] mandatory error condition is added.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

38324 **NAME**

38325 round, roundf, roundl — round to nearest integer value in floating-point format

38326 **SYNOPSIS**

38327
```
#include <math.h>
```

38328
```
double round(double x);
```
38329
```
float roundf(float x);
```
38330
```
long double roundl(long double x);
```

38331 **DESCRIPTION**

38332 CX The functionality described on this reference page is aligned with the ISO C standard. Any
38333 conflict between the requirements described here and the ISO C standard is unintentional. This
38334 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

38335 These functions shall round their argument to the nearest integer value in floating-point format,
38336 rounding halfway cases away from zero, regardless of the current rounding direction.

38337 An application wishing to check for error situations should set *errno* to zero and call
38338 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
38339 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
38340 zero, an error has occurred.

38341 **RETURN VALUE**

38342 Upon successful completion, these functions shall return the rounded integer value.

38343 MX If *x* is NaN, a NaN shall be returned.

38344 If *x* is ±0, or ±Inf, *x* shall be returned.

38345 XSI If the correct value would cause overflow, a range error shall occur and *round*(), *roundf*(), and
38346 *roundl*() shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
38347 respectively.

38348 **ERRORS**

38349 These functions may fail if:

38350 XSI

Range Error	The result overflows.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

38355 **EXAMPLES**

38356 None.

38357 **APPLICATION USAGE**

38358 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
38359 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

38360 **RATIONALE**

38361 None.

38362 **FUTURE DIRECTIONS**

38363 None.

38364 **SEE ALSO**

38365 *feclearexcept*( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, |
38366 Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

38367 **CHANGE HISTORY**

38368 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

38369 **NAME**

38370 scalb — load exponent of a radix-independent floating-point number

38371 **SYNOPSIS**

38372 OB XSI
```
#include <math.h>

double scalb(double x, double n);
```
38373

38374

38375 **DESCRIPTION**

38376 The *scalb*( ) function shall compute $x*r^n$, where r is the radix of the machine's floating-point |
38377 arithmetic. When r is 2, *scalb*( ) shall be equivalent to *ldexp*( ). The value of r is FLT_RADIX |
38378 which is defined in **<float.h>**.

38379 An application wishing to check for error situations should set *errno* to zero and call
38380 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
38381 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
38382 zero, an error has occurred.

38383 **RETURN VALUE**

38384 Upon successful completion, the *scalb*( ) function shall return $x*r^n$. |

38385 If x or n is NaN, a NaN shall be returned.

38386 If n is zero, x shall be returned.

38387 If x is ±Inf and n is not –Inf, x shall be returned.

38388 If x is ±0 and n is not +Inf, x shall be returned.

38389 If x is ±0 and n is +Inf, a domain error shall occur, and either a NaN (if supported), or an
38390 implementation-defined value shall be returned.

38391 If x is ±Inf and n is –Inf, a domain error shall occur, and either a NaN (if supported), or an
38392 implementation-defined value shall be returned.

38393 If the result would cause an overflow, a range error shall occur and ±HUGE_VAL (according to
38394 the sign of x) shall be returned.

38395 If the correct value would cause underflow, and is representable, a range error may occur and
38396 the correct value shall be returned.

38397 If the correct value would cause underflow, and is not representable, a range error may occur,
38398 and 0.0 shall be returned.

38399 **ERRORS**

38400 The *scalb*( ) function shall fail if:

38401 Domain Error — If x is zero and n is +Inf, or x is Inf and n is –Inf.

38402 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
38403 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling |
38404 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception |
38405 shall be raised. |

38406 Range Error — The result would overflow.

38407 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
38408 then *errno* shall be set to [ERANGE]. If the integer expression |
38409 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow |
38410 floating-point exception shall be raised. |

The *scalb*( ) function may fail if:

Range Error — The result underflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

Applications should use either *scalbln*( ), *scalblnf*( ), or *scalblnl*( ) in preference to this function.

IEEE Std 1003.1-200x only defines the behavior for the *scalb*( ) function when the *n* argument is an integer, a NaN, or Inf. The behavior of other values for the *n* argument is unspecified.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *ilogb*( ), *ldexp*( ), *logb*( ), *scalbln*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<float.h>**, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

This function is marked obsolescent.

Although this function is not part of the ISO/IEC 9899:1999 standard, the RETURN VALUE and ERROR sections are updated to align with the error handling in ISO/IEC 9899:1999 standard.

38443 **NAME**

38444 scalbln, scalblnf, scalblnl, scalbn, scalbnf, scalbnl, — compute exponent using FLT_RADIX

38445 **SYNOPSIS**

38446 ```
#include <math.h>

38447 double scalbln(double x, long n);
38448 float scalblnf(float x, long n);
38449 long double scalblnl(long double x, long n);
38450 double scalbn(double x, int n);
38451 float scalbnf(float x, int n);
38452 long double scalbnl(long double x, int n);
```

38453 **DESCRIPTION**

38454 CX The functionality described on this reference page is aligned with the ISO C standard. Any
38455 conflict between the requirements described here and the ISO C standard is unintentional. This
38456 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

38457 These functions shall compute $x * \text{FLT\_RADIX}^n$ efficiently, not normally by computing
38458 $\text{FLT\_RADIX}^n$ explicitly.

38459 An application wishing to check for error situations should set *errno* to zero and call
38460 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
38461 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
38462 zero, an error has occurred.

38463 **RETURN VALUE**

38464 Upon successful completion, these functions shall return $x * \text{FLT\_RADIX}^n$.

38465 If the result would cause overflow, a range error shall occur and these functions shall return
38466 ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (according to the sign of *x*) as appropriate for
38467 the return type of the function.

38468 If the correct value would cause underflow, and is not representable, a range error may occur,
38469 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

38470 MX If *x* is NaN, a NaN shall be returned.

38471 If *x* is ±0, or ±Inf, *x* shall be returned.

38472 If *n* is 0, *x* shall be returned.

38473 If the correct value would cause underflow, and is representable, a range error may occur and
38474 the correct value shall be returned.

38475 **ERRORS**

38476 These functions shall fail if:

38477 Range Error The result overflows.

38478 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
38479 then *errno* shall be set to [ERANGE]. If the integer expression |
38480 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow |
38481 floating-point exception shall be raised. |

38482 These functions may fail if:

38483 Range Error The result underflows.

38484 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
38485 then *errno* shall be set to [ERANGE]. If the integer expression |
```

(math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

These functions are named so as to avoid conflicting with the historical definition of the *scalb*( ) function from the Single UNIX Specification. The difference is that the *scalb*( ) function has second argument of **double** instead of **int**. The *scalb*( ) function is not part of ISO C standard.

The three functions whose second type is **long** are provided because the factor required to scale from the smallest positive floating-point value to the largest finite one, on many implementations, is too large to represent in the minimum-width **int** format.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *scalb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

38507 **NAME**

38508 scalbn, scalbnf, scalbnl, — compute exponent using FLT_RADIX

38509 **SYNOPSIS**

38510 `#include <math.h>`

38511 `double scalbn(double x, int n);`
38512 `float scalbnf(float x, int n);`
38513 `long double scalbnl(long double x, int n);`

38514 **DESCRIPTION**

38515 Refer to *scalbln*( ).

38516 **NAME**

38517 scanf — convert formatted input

38518 **SYNOPSIS**

38519 `#include <stdio.h>`

38520 `int scanf(const char *restrict format, ... );`

38521 **DESCRIPTION**

38522 Refer to *fscanf*( ).

38523 **NAME**

38524 sched_get_priority_max, sched_get_priority_min — get priority limits (**REALTIME**)

38525 **SYNOPSIS**

38526 PS
```
#include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```
38529

38530 **DESCRIPTION**

38531 The *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions shall return the appropriate
38532 maximum or minimum, respectively, for the scheduling policy specified by *policy*.

38533 The value of *policy* shall be one of the scheduling policy values defined in **<sched.h>**.

38534 **RETURN VALUE**

38535 If successful, the *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions shall return the
38536 appropriate maximum or minimum values, respectively. If unsuccessful, they shall return a
38537 value of –1 and set *errno* to indicate the error.

38538 **ERRORS**

38539 The *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions shall fail if:

38540 [EINVAL] The value of the *policy* parameter does not represent a defined scheduling
38541 policy.

38542 **EXAMPLES**

38543 None.

38544 **APPLICATION USAGE**

38545 None.

38546 **RATIONALE**

38547 None.

38548 **FUTURE DIRECTIONS**

38549 None.

38550 **SEE ALSO**

38551 *sched_getparam*( ), *sched_setparam*( ), *sched_getscheduler*( ), *sched_rr_get_interval*( ),
38552 *sched_setscheduler*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**

38553 **CHANGE HISTORY**

38554 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38555 **Issue 6**

38556 These functions are marked as part of the Process Scheduling option.

38557 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38558 implementation does not support the Process Scheduling option.

38559 The [ESRCH] error condition has been removed since these functions do not take a *pid*
38560 argument.

38561 **NAME**

38562 sched_getparam — get scheduling parameters (**REALTIME**)

38563 **SYNOPSIS**

38564 PS
```
#include <sched.h>

int sched_getparam(pid_t pid, struct sched_param *param);
```
38565

38566

38567 **DESCRIPTION**

38568 The *sched_getparam*( ) function shall return the scheduling parameters of a process specified by
38569 *pid* in the **sched_param** structure pointed to by *param*.

38570 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
38571 parameters for the process whose process ID is equal to *pid* shall be returned.

38572 If *pid* is zero, the scheduling parameters for the calling process shall be returned. The behavior of
38573 the *sched_getparam*( ) function is unspecified if the value of *pid* is negative.

38574 **RETURN VALUE**

38575 Upon successful completion, the *sched_getparam*( ) function shall return zero. If the call to
38576 *sched_getparam*( ) is unsuccessful, the function shall return a value of –1 and set *errno* to indicate
38577 the error.

38578 **ERRORS**

38579 The *sched_getparam*( ) function shall fail if:

38580 [EPERM] The requesting process does not have permission to obtain the scheduling
38581 parameters of the specified process.

38582 [ESRCH] No process can be found corresponding to that specified by *pid*.

38583 **EXAMPLES**

38584 None.

38585 **APPLICATION USAGE**

38586 None.

38587 **RATIONALE**

38588 None.

38589 **FUTURE DIRECTIONS**

38590 None.

38591 **SEE ALSO**

38592 *sched_getscheduler*( ), *sched_setparam*( ), *sched_setscheduler*( ), the Base Definitions volume of
38593 IEEE Std 1003.1-200x, **<sched.h>**

38594 **CHANGE HISTORY**

38595 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38596 **Issue 6**

38597 The *sched_getparam*( ) function is marked as part of the Process Scheduling option.

38598 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38599 implementation does not support the Process Scheduling option.

38600 **NAME**

38601 sched_getscheduler — get scheduling policy (**REALTIME**)

38602 **SYNOPSIS**

38603 PS
```
#include <sched.h>
```

38604
```
int sched_getscheduler(pid_t pid);
```

38605

38606 **DESCRIPTION**

38607 The *sched_getscheduler*( ) function shall return the scheduling policy of the process specified by
38608 *pid*. If the value of *pid* is negative, the behavior of the *sched_getscheduler*( ) function is
38609 unspecified.

38610 The values that can be returned by *sched_getscheduler*( ) are defined in the **<sched.h>** header. |

38611 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
38612 policy shall be returned for the process whose process ID is equal to *pid*.

38613 If *pid* is zero, the scheduling policy shall be returned for the calling process.

38614 **RETURN VALUE**

38615 Upon successful completion, the *sched_getscheduler*( ) function shall return the scheduling policy
38616 of the specified process. If unsuccessful, the function shall return –1 and set *errno* to indicate the
38617 error.

38618 **ERRORS**

38619 The *sched_getscheduler*( ) function shall fail if:

38620 [EPERM] The requesting process does not have permission to determine the scheduling
38621 policy of the specified process.

38622 [ESRCH] No process can be found corresponding to that specified by *pid*.

38623 **EXAMPLES**

38624 None.

38625 **APPLICATION USAGE**

38626 None.

38627 **RATIONALE**

38628 None.

38629 **FUTURE DIRECTIONS**

38630 None.

38631 **SEE ALSO**

38632 *sched_getparam*( ), *sched_setparam*( ), *sched_setscheduler*( ), the Base Definitions volume of
38633 IEEE Std 1003.1-200x, **<sched.h>**

38634 **CHANGE HISTORY**

38635 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38636 **Issue 6**

38637 The *sched_getscheduler*( ) function is marked as part of the Process Scheduling option.

38638 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38639 implementation does not support the Process Scheduling option.

38640 **NAME**

38641 sched_rr_get_interval — get execution time limits (**REALTIME**)

38642 **SYNOPSIS**

38643 PS
```
#include <sched.h>

int sched_rr_get_interval(pid_t pid, struct timespec *interval);
```
38644

38645

38646 **DESCRIPTION**

38647 The *sched_rr_get_interval*( ) function shall update the **timespec** structure referenced by the
38648 *interval* argument to contain the current execution time limit (that is, time quantum) for the
38649 process specified by *pid*. If *pid* is zero, the current execution time limit for the calling process
38650 shall be returned.

38651 **RETURN VALUE**

38652 If successful, the *sched_rr_get_interval*( ) function shall return zero. Otherwise, it shall return a
38653 value of –1 and set *errno* to indicate the error.

38654 **ERRORS**

38655 The *sched_rr_get_interval*( ) function shall fail if:

38656 [ESRCH] No process can be found corresponding to that specified by *pid*.

38657 **EXAMPLES**

38658 None.

38659 **APPLICATION USAGE**

38660 None.

38661 **RATIONALE**

38662 None.

38663 **FUTURE DIRECTIONS**

38664 None.

38665 **SEE ALSO**

38666 *sched_getparam*( ), *sched_get_priority_max*( ), *sched_getscheduler*( ), *sched_setparam*( ),
38667 *sched_setscheduler*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**

38668 **CHANGE HISTORY**

38669 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38670 **Issue 6**

38671 The *sched_rr_get_interval*( ) function is marked as part of the Process Scheduling option.

38672 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38673 implementation does not support the Process Scheduling option.

38674 **NAME**

38675 sched_setparam — set scheduling parameters (**REALTIME**)

38676 **SYNOPSIS**

38677 PS
```
#include <sched.h>

int sched_setparam(pid_t pid, const struct sched_param *param);
```
38678

38679

38680 **DESCRIPTION**

38681 The *sched_setparam*( ) function shall set the scheduling parameters of the process specified by *pid*
38682 to the values specified by the **sched_param** structure pointed to by *param*. The value of the
38683 *sched_priority* member in the **sched_param** structure shall be any integer within the inclusive
38684 priority range for the current scheduling policy of the process specified by *pid*. Higher
38685 numerical values for the priority represent higher priorities. If the value of *pid* is negative, the
38686 behavior of the *sched_setparam*( ) function is unspecified.

38687 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
38688 parameters shall be set for the process whose process ID is equal to *pid*.

38689 If *pid* is zero, the scheduling parameters shall be set for the calling process.

38690 The conditions under which one process has permission to change the scheduling parameters of
38691 another process are implementation-defined.

38692 Implementations may require the requesting process to have the appropriate privilege to set its
38693 own scheduling parameters or those of another process.

38694 The target process, whether it is running or not running, shall be moved to the tail of the thread
38695 list for its priority.

38696 If the priority of the process specified by the *pid* argument is set higher than that of the lowest
38697 priority running process and if the specified process is ready to run, the process specified by the
38698 *pid* argument shall preempt a lowest priority running process. Similarly, if the process calling |
38699 *sched_setparam*( ) sets its own priority lower than that of one or more other non-empty process
38700 lists, then the process that is the head of the highest priority list shall also preempt the calling |
38701 process. Thus, in either case, the originating process might not receive notification of the |
38702 completion of the requested priority change until the higher priority process has executed.

38703 SS If the scheduling policy of the target process is SCHED_SPORADIC, the value specified by the
38704 *sched_ss_low_priority* member of the *param* argument shall be any integer within the inclusive
38705 priority range for the sporadic server policy. The *sched_ss_repl_period* and *sched_ss_init_budget*
38706 members of the *param* argument shall represent the time parameters to be used by the sporadic
38707 server scheduling policy for the target process. The *sched_ss_max_repl* member of the *param*
38708 argument shall represent the maximum number of replenishments that are allowed to be
38709 pending simultaneously for the process scheduled under this scheduling policy.

38710 The specified *sched_ss_repl_period* shall be greater than or equal to the specified
38711 *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.

38712 The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the
38713 function to succeed; if not, the function shall fail.

38714 If the scheduling policy of the target process is either SCHED_FIFO or SCHED_RR, the
38715 *sched_ss_low_priority*, *sched_ss_repl_period*, and *sched_ss_init_budget* members of the *param*
38716 argument shall have no effect on the scheduling behavior. If the scheduling policy of this process
38717 is not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, the effects of these members are
38718 implementation-defined; this case includes the SCHED_OTHER policy.

If the current scheduling policy for the process specified by *pid* is not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, the result is implementation-defined; this case includes the SCHED_OTHER policy.

The effect of this function on individual threads is dependent on the scheduling contention scope of the threads:

- For threads with system scheduling contention scope, these functions shall have no effect on their scheduling.
- For threads with process scheduling contention scope, the threads' scheduling parameters shall not be affected. However, the scheduling of these threads with respect to threads in other processes may be dependent on the scheduling parameters of their process, which are governed using these functions.

If an implementation supports a two-level scheduling model in which library threads are multiplexed on top of several kernel-scheduled entities, then the underlying kernel-scheduled entities for the system contention scope threads shall not be affected by these functions.

The underlying kernel-scheduled entities for the process contention scope threads shall have their scheduling parameters changed to the value specified in *param*. Kernel scheduled entities for use by process contention scope threads that are created after this call completes shall inherit their scheduling policy and associated scheduling parameters from the process.

This function is not atomic with respect to other threads in the process. Threads may continue to execute while this function call is in the process of changing the scheduling policy for the underlying kernel-scheduled entities used by the process contention scope threads.

RETURN VALUE

If successful, the *sched_setparam*( ) function shall return zero.

If the call to *sched_setparam*( ) is unsuccessful, the priority shall remain unchanged, and the function shall return a value of –1 and set *errno* to indicate the error.

ERRORS

The *sched_setparam*( ) function shall fail if:

[EINVAL] One or more of the requested scheduling parameters is outside the range defined for the scheduling policy of the specified *pid*.

[EPERM] The requesting process does not have permission to set the scheduling parameters for the specified process, or does not have the appropriate privilege to invoke *sched_setparam*( ).

[ESRCH] No process can be found corresponding to that specified by *pid*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

38760 **SEE ALSO**

38761 *sched_getparam*( ), *sched_getscheduler*( ), *sched_setscheduler*( ), the Base Definitions volume of
38762 IEEE Std 1003.1-200x, **<sched.h>**

38763 **CHANGE HISTORY**

38764 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38765 **Issue 6**

38766 The *sched_setparam*( ) function is marked as part of the Process Scheduling option.

38767 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38768 implementation does not support the Process Scheduling option.

38769 The following new requirements on POSIX implementations derive from alignment with the
38770 Single UNIX Specification:

38771 • In the DESCRIPTION, the effect of this function on a thread's scheduling parameters is
38772 added.

38773 • Sections describing two-level scheduling and atomicity of the function are added.

38774 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

38775 IEEE PASC Interpretation 1003.1 #100 is applied.

38776 **NAME**

38777 sched_setscheduler — set scheduling policy and parameters (**REALTIME**)

38778 **SYNOPSIS**

38779 PS
```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy,
    const struct sched_param *param);
```
38782

38783 **DESCRIPTION**

38784 The *sched_setscheduler*( ) function shall set the scheduling policy and scheduling parameters of
38785 the process specified by *pid* to *policy* and the parameters specified in the **sched_param** structure
38786 pointed to by *param*, respectively. The value of the *sched_priority* member in the **sched_param**
38787 structure shall be any integer within the inclusive priority range for the scheduling policy
38788 specified by *policy*. If the value of *pid* is negative, the behavior of the *sched_setscheduler*( )
38789 function is unspecified.

38790 The possible values for the *policy* parameter are defined in the **<sched.h>** header. |

38791 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
38792 policy and scheduling parameters shall be set for the process whose process ID is equal to *pid*.

38793 If *pid* is zero, the scheduling policy and scheduling parameters shall be set for the calling
38794 process.

38795 The conditions under which one process has the appropriate privilege to change the scheduling
38796 parameters of another process are implementation-defined.

38797 Implementations may require that the requesting process have permission to set its own
38798 scheduling parameters or those of another process. Additionally, implementation-defined
38799 restrictions may apply as to the appropriate privileges required to set a process' own scheduling
38800 policy, or another process' scheduling policy, to a particular value.

38801 The *sched_setscheduler*( ) function shall be considered successful if it succeeds in setting the |
38802 scheduling policy and scheduling parameters of the process specified by *pid* to the values
38803 specified by *policy* and the structure pointed to by *param*, respectively.

38804 SS If the scheduling policy specified by *policy* is SCHED_SPORADIC, the value specified by the
38805 *sched_ss_low_priority* member of the *param* argument shall be any integer within the inclusive
38806 priority range for the sporadic server policy. The *sched_ss_repl_period* and *sched_ss_init_budget*
38807 members of the *param* argument shall represent the time parameters used by the sporadic server
38808 scheduling policy for the target process. The *sched_ss_max_repl* member of the *param* argument
38809 shall represent the maximum number of replenishments that are allowed to be pending
38810 simultaneously for the process scheduled under this scheduling policy.

38811 The specified *sched_ss_repl_period* shall be greater than or equal to the specified
38812 *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.

38813 The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the
38814 function to succeed; if not, the function shall fail.

38815 If the scheduling policy specified by *policy* is either SCHED_FIFO or SCHED_RR, the
38816 *sched_ss_low_priority*, *sched_ss_repl_period*, and *sched_ss_init_budget* members of the *param*
38817 argument shall have no effect on the scheduling behavior.

38818 The effect of this function on individual threads is dependent on the scheduling contention
38819 scope of the threads:

- For threads with system scheduling contention scope, these functions shall have no effect on their scheduling.
- For threads with process scheduling contention scope, the threads' scheduling policy and associated parameters shall not be affected. However, the scheduling of these threads with respect to threads in other processes may be dependent on the scheduling parameters of their process, which are governed using these functions.

If an implementation supports a two-level scheduling model in which library threads are multiplexed on top of several kernel-scheduled entities, then the underlying kernel-scheduled entities for the system contention scope threads shall not be affected by these functions.

The underlying kernel-scheduled entities for the process contention scope threads shall have their scheduling policy and associated scheduling parameters changed to the values specified in *policy* and *param*, respectively. Kernel scheduled entities for use by process contention scope threads that are created after this call completes shall inherit their scheduling policy and associated scheduling parameters from the process.

This function is not atomic with respect to other threads in the process. Threads may continue to execute while this function call is in the process of changing the scheduling policy and associated scheduling parameters for the underlying kernel-scheduled entities used by the process contention scope threads.

RETURN VALUE

Upon successful completion, the function shall return the former scheduling policy of the specified process. If the *sched_setscheduler*( ) function fails to complete successfully, the policy and scheduling paramenters shall remain unchanged, and the function shall return a value of –1 and set *errno* to indicate the error.

ERRORS

The *sched_setscheduler*( ) function shall fail if:

[EINVAL] The value of the *policy* parameter is invalid, or one or more of the parameters contained in *param* is outside the valid range for the specified scheduling policy.

[EPERM] The requesting process does not have permission to set either or both of the scheduling parameters or the scheduling policy of the specified process.

[ESRCH] No process can be found corresponding to that specified by *pid*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sched_getparam( ), *sched_getscheduler*( ), *sched_setparam*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**

38862 **CHANGE HISTORY**

38863 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38864 **Issue 6**

38865 The *sched_setscheduler*( ) function is marked as part of the Process Scheduling option.

38866 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38867 implementation does not support the Process Scheduling option.

38868 The following new requirements on POSIX implementations derive from alignment with the
38869 Single UNIX Specification:

38870 • In the DESCRIPTION, the effect of this function on a thread's scheduling parameters is
38871 added.

38872 • Sections describing two-level scheduling and atomicity of the function are added.

38873 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

38874 **NAME**

38875 sched_yield — yield processor

38876 **SYNOPSIS**

38877 PS|THR
```
#include <sched.h>
```
38878
```
int sched_yield(void);
```
38879

38880 **DESCRIPTION**

38881 The *sched_yield*( ) function shall force the running thread to relinquish the processor until it again
38882 becomes the head of its thread list. It takes no arguments.

38883 **RETURN VALUE**

38884 The *sched_yield*( ) function shall return 0 if it completes successfully; otherwise, it shall return a
38885 value of –1 and set *errno* to indicate the error.

38886 **ERRORS**

38887 No errors are defined.

38888 **EXAMPLES**

38889 None.

38890 **APPLICATION USAGE**

38891 None.

38892 **RATIONALE**

38893 None.

38894 **FUTURE DIRECTIONS**

38895 None.

38896 **SEE ALSO**

38897 The Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**

38898 **CHANGE HISTORY**

38899 First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the
38900 POSIX Threads Extension.

38901 **Issue 6**

38902 The *sched_yield*( ) function is now marked as part of the Process Scheduling and Threads options.

38903 **NAME**

38904 seed48 — seed uniformly distributed pseudo-random non-negative long integer generator

38905 **SYNOPSIS**

38906 XSI
```
#include <stdlib.h>
```

38907
```
unsigned short *seed48(unsigned short seed16v[3]);
```

38908

38909 **DESCRIPTION**

38910 Refer to *drand48*( ).

38911 **NAME**

38912 seekdir — set position of directory stream

38913 **SYNOPSIS**

38914 XSI
```
#include <dirent.h>

void seekdir(DIR *dirp, long loc);
```
38915

38916

38917 **DESCRIPTION**

38918 The *seekdir*( ) function shall set the position of the next *readdir*( ) operation on the directory
38919 stream specified by *dirp* to the position specified by *loc*. The value of *loc* should have been
38920 returned from an earlier call to *telldir*( ). The new position reverts to the one associated with the
38921 directory stream when *telldir*( ) was performed.

38922 If the value of *loc* was not obtained from an earlier call to *telldir*( ), or if a call to *rewinddir*( )
38923 occurred between the call to *telldir*( ) and the call to *seekdir*( ), the results of subsequent calls to
38924 *readdir*( ) are unspecified.

38925 **RETURN VALUE**

38926 The *seekdir*( ) function shall not return a value.

38927 **ERRORS**

38928 No errors are defined.

38929 **EXAMPLES**

38930 None.

38931 **APPLICATION USAGE**

38932 None.

38933 **RATIONALE**

38934 The original standard developers perceived that there were restrictions on the use of the
38935 *seekdir*( ) and *telldir*( ) functions related to implementation details, and for that reason these
38936 functions need not be supported on all POSIX-conforming systems. They are required on
38937 implementations supporting the XSI extension.

38938 One of the perceived problems of implementation is that returning to a given point in a directory
38939 is quite difficult to describe formally, in spite of its intuitive appeal, when systems that use B-
38940 trees, hashing functions, or other similar mechanisms to order their directories are considered.
38941 The definition of *seekdir*( ) and *telldir*( ) does not specify whether, when using these interfaces, a
38942 given directory entry will be seen at all, or more than once.

38943 On systems not supporting these functions, their capability can sometimes be accomplished by
38944 saving a filename found by *readdir*( ) and later using *rewinddir*( ) and a loop on *readdir*( ) to
38945 relocate the position from which the filename was saved.

38946 **FUTURE DIRECTIONS**

38947 None.

38948 **SEE ALSO**

38949 *opendir*( ), *readdir*( ), *telldir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**,
38950 **<stdio.h>**, **<sys/types.h>**

38951 **CHANGE HISTORY**

38952 First released in Issue 2.

38953 **Issue 6**

38954 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

38955 **NAME**

38956 select — synchronous I/O multiplexing

38957 **SYNOPSIS**

38958 `#include <sys/time.h>`

38959 `int select(int nfds, fd_set *restrict readfds,`
38960 `fd_set *restrict writefds, fd_set *restrict errorfds,`
38961 `struct timeval *restrict timeout);`

38962

38963 **DESCRIPTION**

38964 Refer to *pselect*( ).

38965 **NAME**

38966 sem_close — close a named semaphore (**REALTIME**)

38967 **SYNOPSIS**

38968 SEM
```
#include <semaphore.h>

int sem_close(sem_t *sem);
```
38969

38970

38971 **DESCRIPTION**

38972 The *sem_close*( ) function shall indicate that the calling process is finished using the named
38973 semaphore indicated by *sem*. The effects of calling *sem_close*( ) for an unnamed semaphore (one
38974 created by *sem_init*( )) are undefined. The *sem_close*( ) function shall deallocate (that is, make
38975 available for reuse by a subsequent *sem_open*( ) by this process) any system resources allocated
38976 by the system for use by this process for this semaphore. The effect of subsequent use of the
38977 semaphore indicated by *sem* by this process is undefined. If the semaphore has not been
38978 removed with a successful call to *sem_unlink*( ), then *sem_close*( ) has no effect on the state of the
38979 semaphore. If the *sem_unlink*( ) function has been successfully invoked for *name* after the most
38980 recent call to *sem_open*( ) with O_CREAT for this semaphore, then when all processes that have
38981 opened the semaphore close it, the semaphore is no longer accessible.

38982 **RETURN VALUE**

38983 Upon successful completion, a value of zero shall be returned. Otherwise, a value of –1 shall be
38984 returned and *errno* set to indicate the error.

38985 **ERRORS**

38986 The *sem_close*( ) function shall fail if:

38987 [EINVAL] The *sem* argument is not a valid semaphore descriptor.

38988 **EXAMPLES**

38989 None.

38990 **APPLICATION USAGE**

38991 The *sem_close*( ) function is part of the Semaphores option and need not be available on all
38992 implementations.

38993 **RATIONALE**

38994 None.

38995 **FUTURE DIRECTIONS**

38996 None.

38997 **SEE ALSO**

38998 *semctl*( ), *semget*( ), *semop*( ), *sem_init*( ), *sem_open*( ), *sem_unlink*( ), the Base Definitions volume of
38999 IEEE Std 1003.1-200x, **<semaphore.h>**

39000 **CHANGE HISTORY**

39001 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39002 **Issue 6**

39003 The *sem_close*( ) function is marked as part of the Semaphores option.

39004 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39005 implementation does not support the Semaphores option.

39006 **NAME**

39007 sem_destroy — destroy an unnamed semaphore (**REALTIME**)

39008 **SYNOPSIS**

39009 SEM
```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```
39010

39011

39012 **DESCRIPTION**

39013 The *sem_destroy*( ) function shall destroy the unnamed semaphore indicated by *sem*. Only a
39014 semaphore that was created using *sem_init*( ) may be destroyed using *sem_destroy*( ); the effect of
39015 calling *sem_destroy*( ) with a named semaphore is undefined. The effect of subsequent use of the
39016 semaphore *sem* is undefined until *sem* is reinitialized by another call to *sem_init*( ).

39017 It is safe to destroy an initialized semaphore upon which no threads are currently blocked. The
39018 effect of destroying a semaphore upon which other threads are currently blocked is undefined.

39019 **RETURN VALUE**

39020 Upon successful completion, a value of zero shall be returned. Otherwise, a value of –1 shall be
39021 returned and *errno* set to indicate the error.

39022 **ERRORS**

39023 The *sem_destroy*( ) function shall fail if:

39024 [EINVAL] The *sem* argument is not a valid semaphore.

39025 The *sem_destroy*( ) function may fail if:

39026 [EBUSY] There are currently processes blocked on the semaphore.

39027 **EXAMPLES**

39028 None.

39029 **APPLICATION USAGE**

39030 The *sem_destroy*( ) function is part of the Semaphores option and need not be available on all
39031 implementations.

39032 **RATIONALE**

39033 None.

39034 **FUTURE DIRECTIONS**

39035 None.

39036 **SEE ALSO**

39037 *semctl*( ), *semget*( ), *semop*( ), *sem_init*( ), *sem_open*( ), the Base Definitions volume of
39038 IEEE Std 1003.1-200x, **<semaphore.h>**

39039 **CHANGE HISTORY**

39040 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39041 **Issue 6**

39042 The *sem_destroy*( ) function is marked as part of the Semaphores option.

39043 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39044 implementation does not support the Semaphores option.

39045 **NAME**

39046 sem_getvalue — get the value of a semaphore (**REALTIME**)

39047 **SYNOPSIS**

39048 SEM
```
#include <semaphore.h>

int sem_getvalue(sem_t *restrict sem, int *restrict sval);
```
39049

39050

39051 **DESCRIPTION**

39052 The *sem_getvalue*( ) function shall update the location referenced by the *sval* argument to have
39053 the value of the semaphore referenced by *sem* without affecting the state of the semaphore. The
39054 updated value represents an actual semaphore value that occurred at some unspecified time
39055 during the call, but it need not be the actual value of the semaphore when it is returned to the
39056 calling process.

39057 If *sem* is locked, then the value returned by *sem_getvalue*( ) is either zero or a negative number
39058 whose absolute value represents the number of processes waiting for the semaphore at some
39059 unspecified time during the call.

39060 **RETURN VALUE**

39061 Upon successful completion, the *sem_getvalue*( ) function shall return a value of zero. Otherwise,
39062 it shall return a value of –1 and set *errno* to indicate the error.

39063 **ERRORS**

39064 The *sem_getvalue*( ) function shall fail if:

39065 [EINVAL] The *sem* argument does not refer to a valid semaphore.

39066 **EXAMPLES**

39067 None.

39068 **APPLICATION USAGE**

39069 The *sem_getvalue*( ) function is part of the Semaphores option and need not be available on all
39070 implementations.

39071 **RATIONALE**

39072 None.

39073 **FUTURE DIRECTIONS**

39074 None.

39075 **SEE ALSO**

39076 *semctl*( ), *semget*( ), *semop*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_wait*( ), the Base
39077 Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**

39078 **CHANGE HISTORY**

39079 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39080 **Issue 6**

39081 The *sem_getvalue*( ) function is marked as part of the Semaphores option.

39082 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39083 implementation does not support the Semaphores option.

39084 The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with
39085 IEEE Std 1003.1d-1999.

39086 The **restrict** keyword is added to the *sem_getvalue*( ) prototype for alignment with the
39087 ISO / IEC 9899: 1999 standard.

39088 **NAME**

39089 sem_init — initialize an unnamed semaphore (**REALTIME**)

39090 **SYNOPSIS**

39091 SEM
```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned value);
```
39092

39093

39094 **DESCRIPTION**

39095 The *sem_init*( ) function shall initialize the unnamed semaphore referred to by *sem*. The value of |
39096 the initialized semaphore shall be *value*. Following a successful call to *sem_init*( ), the semaphore
39097 may be used in subsequent calls to *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and *sem_destroy*( ).
39098 This semaphore shall remain usable until the semaphore is destroyed.

39099 If the *pshared* argument has a non-zero value, then the semaphore is shared between processes;
39100 in this case, any process that can access the semaphore *sem* can use *sem* for performing
39101 *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and *sem_destroy*( ) operations.

39102 Only *sem* itself may be used for performing synchronization. The result of referring to copies of
39103 *sem* in calls to *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and *sem_destroy*( ), is undefined.

39104 If the *pshared* argument is zero, then the semaphore is shared between threads of the process; any
39105 thread in this process can use *sem* for performing *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and
39106 *sem_destroy*( ) operations. The use of the semaphore by threads other than those created in the
39107 same process is undefined.

39108 Attempting to initialize an already initialized semaphore results in undefined behavior.

39109 **RETURN VALUE**

39110 Upon successful completion, the *sem_init*( ) function shall initialize the semaphore in *sem*.
39111 Otherwise, it shall return –1 and set *errno* to indicate the error.

39112 **ERRORS**

39113 The *sem_init*( ) function shall fail if:

39114 [EINVAL] The *value* argument exceeds {SEM_VALUE_MAX}.

39115 [ENOSPC] A resource required to initialize the semaphore has been exhausted, or the
39116 limit on semaphores ({SEM_NSEMS_MAX}) has been reached.

39117 [EPERM] The process lacks the appropriate privileges to initialize the semaphore.

39118 **EXAMPLES**

39119 None.

39120 **APPLICATION USAGE**

39121 The *sem_init*( ) function is part of the Semaphores option and need not be available on all
39122 implementations.

39123 **RATIONALE**

39124 Although this volume of IEEE Std 1003.1-200x fails to specify a successful return value, it is
39125 likely that a later version may require the implementation to return a value of zero if the call to
39126 *sem_init*( ) is successful.

39127 **FUTURE DIRECTIONS**

39128 None.

39129 **SEE ALSO**

39130 *sem_destroy*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_wait*( ), the Base Definitions
39131 volume of IEEE Std 1003.1-200x, **<semaphore.h>**

39132 **CHANGE HISTORY**

39133 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39134 **Issue 6**

39135 The *sem_init*( ) function is marked as part of the Semaphores option.

39136 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39137 implementation does not support the Semaphores option.

39138 The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with
39139 IEEE Std 1003.1d-1999.

NAME

sem_open — initialize and open a named semaphore (**REALTIME**)

SYNOPSIS

SEM

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...);
```

DESCRIPTION

The *sem_open*( ) function shall establish a connection between a named semaphore and a process. Following a call to *sem_open*( ) with semaphore name *name*, the process may reference the semaphore associated with *name* using the address returned from the call. This semaphore may be used in subsequent calls to *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and *sem_close*( ). The semaphore remains usable by this process until the semaphore is closed by a successful call to *sem_close*( ), *_exit*( ), or one of the *exec* functions.

The *oflag* argument controls whether the semaphore is created or merely accessed by the call to *sem_open*( ). The following flag bits may be set in *oflag*:

O_CREAT This flag is used to create a semaphore if it does not already exist. If O_CREAT is set and the semaphore already exists, then O_CREAT has no effect, except as noted under O_EXCL. Otherwise, *sem_open*( ) creates a named semaphore. The O_CREAT flag requires a third and a fourth argument: *mode*, which is of type **mode_t**, and *value*, which is of type **unsigned**. The semaphore is created with an initial value of *value*. Valid initial values for semaphores are less than or equal to {SEM_VALUE_MAX}.

The user ID of the semaphore is set to the effective user ID of the process; the group ID of the semaphore is set to a system default group ID or to the effective group ID of the process. The permission bits of the semaphore are set to the value of the *mode* argument except those set in the file mode creation mask of the process. When bits in *mode* other than the file permission bits are specified, the effect is unspecified.

After the semaphore named *name* has been created by *sem_open*( ) with the O_CREAT flag, other processes can connect to the semaphore by calling *sem_open*( ) with the same value of *name*.

O_EXCL If O_EXCL and O_CREAT are set, *sem_open*( ) fails if the semaphore *name* exists. The check for the existence of the semaphore and the creation of the semaphore if it does not exist are atomic with respect to other processes executing *sem_open*( ) with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the effect is undefined.

If flags other than O_CREAT and O_EXCL are specified in the *oflag* parameter, the effect is unspecified.

The *name* argument points to a string naming a semaphore object. It is unspecified whether the name appears in the file system and is visible to functions that take pathnames as arguments. | The *name* argument conforms to the construction rules for a pathname. If *name* begins with the | slash character, then processes calling *sem_open*( ) with the same value of *name* shall refer to the same semaphore object, as long as that name has not been removed. If *name* does not begin with the slash character, the effect is implementation-defined. The interpretation of slash characters other than the leading slash character in *name* is implementation-defined.

If a process makes multiple successful calls to *sem_open*( ) with the same value for *name*, the | same semaphore address shall be returned for each such successful call, provided that there |

have been no calls to *sem_unlink*( ) for this semaphore.

References to copies of the semaphore produce undefined results.

RETURN VALUE

Upon successful completion, the *sem_open*( ) function shall return the address of the semaphore. Otherwise, it shall return a value of SEM_FAILED and set *errno* to indicate the error. The symbol SEM_FAILED is defined in the **<semaphore.h>** header. No successful return from *sem_open*( ) shall return the value SEM_FAILED.

ERRORS

If any of the following conditions occur, the *sem_open*( ) function shall return SEM_FAILED and set *errno* to the corresponding value:

[EACCES] The named semaphore exists and the permissions specified by *oflag* are denied, or the named semaphore does not exist and permission to create the named semaphore is denied.

[EEXIST] O_CREAT and O_EXCL are set and the named semaphore already exists.

[EINTR] The *sem_open*( ) operation was interrupted by a signal.

[EINVAL] The *sem_open*( ) operation is not supported for the given name, or O_CREAT was specified in *oflag* and *value* was greater than {SEM_VALUE_MAX}.

[EMFILE] Too many semaphore descriptors or file descriptors are currently in use by this process.

[ENAMETOOLONG] The length of the *name* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

[ENFILE] Too many semaphores are currently open in the system.

[ENOENT] O_CREAT is not set and the named semaphore does not exist.

[ENOSPC] There is insufficient space for the creation of the new named semaphore.

EXAMPLES

None.

APPLICATION USAGE

The *sem_open*( ) function is part of the Semaphores option and need not be available on all implementations.

RATIONALE

An earlier version of this volume of IEEE Std 1003.1-200x required an error return value of –1 with the type **sem_t *** for the *sem_open*( ) function, which is not guaranteed to be portable across implementations. The revised text provides the symbolic error code SEM_FAILED to eliminate the type conflict.

FUTURE DIRECTIONS

None.

SEE ALSO

semctl( ), *semget*( ), *semop*( ), *sem_close*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_unlink*( ), *sem_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**

39227 **CHANGE HISTORY**

39228 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39229 **Issue 6**

39230 The *sem_open*( ) function is marked as part of the Semaphores option.

39231 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39232 implementation does not support the Semaphores option.

39233 The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with
39234 IEEE Std 1003.1d-1999.

39235 **NAME**

39236 sem_post — unlock a semaphore (**REALTIME**)

39237 **SYNOPSIS**

39238 SEM
```
#include <semaphore.h>
```

39239
```
int sem_post(sem_t *sem);
```

39240

39241 **DESCRIPTION**

39242 The *sem_post*( ) function shall unlock the semaphore referenced by *sem* by performing a
39243 semaphore unlock operation on that semaphore.

39244 If the semaphore value resulting from this operation is positive, then no threads were blocked
39245 waiting for the semaphore to become unlocked; the semaphore value is simply incremented.

39246 If the value of the semaphore resulting from this operation is zero, then one of the threads
39247 blocked waiting for the semaphore shall be allowed to return successfully from its call to
39248 PS *sem_wait*( ). If the Process Scheduling option is supported, the thread to be unblocked shall be
39249 chosen in a manner appropriate to the scheduling policies and parameters in effect for the
39250 blocked threads. In the case of the schedulers SCHED_FIFO and SCHED_RR, the highest
39251 priority waiting thread shall be unblocked, and if there is more than one highest priority thread
39252 blocked waiting for the semaphore, then the highest priority thread that has been waiting the
39253 longest shall be unblocked. If the Process Scheduling option is not defined, the choice of a thread
39254 to unblock is unspecified.

39255 SS If the Process Sporadic Server option is supported, and the scheduling policy is
39256 SCHED_SPORADIC, the semantics are as per SCHED_FIFO above.

39257 The *sem_post*( ) function shall be reentrant with respect to signals and may be invoked from a
39258 signal-catching function.

39259 **RETURN VALUE**

39260 If successful, the *sem_post*( ) function shall return zero; otherwise, the function shall return –1
39261 and set *errno* to indicate the error.

39262 **ERRORS**

39263 The *sem_post*( ) function shall fail if:

39264 [EINVAL] The *sem* argument does not refer to a valid semaphore.

39265 **EXAMPLES**

39266 None.

39267 **APPLICATION USAGE**

39268 The *sem_post*( ) function is part of the Semaphores option and need not be available on all
39269 implementations.

39270 **RATIONALE**

39271 None.

39272 **FUTURE DIRECTIONS**

39273 None.

39274 **SEE ALSO**

39275 *semctl*( ), *semget*( ), *semop*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_wait*( ), the Base Definitions
39276 volume of IEEE Std 1003.1-200x, **<semaphore.h>**

39277 **CHANGE HISTORY**

39278 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39279 **Issue 6**

39280 The *sem_post*( ) function is marked as part of the Semaphores option.

39281 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39282 implementation does not support the Semaphores option.

39283 The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with
39284 IEEE Std 1003.1d-1999.

39285 SCHED_SPORADIC is added to the list of scheduling policies for which the thread that is to be
39286 unblocked is specified for alignment with IEEE Std 1003.1d-1999.

NAME

sem_timedwait — lock a semaphore (**ADVANCED REALTIME**)

SYNOPSIS

SEM TMO
```
#include <semaphore.h>
#include <time.h>

int sem_timedwait(sem_t *restrict sem,
    const struct timespec *restrict abs_timeout);
```

DESCRIPTION

The *sem_timedwait*( ) function shall lock the semaphore referenced by *sem* as in the *sem_wait*( ) function. However, if the semaphore cannot be locked without waiting for another process or thread to unlock the semaphore by performing a *sem_post*( ) function, this wait shall be terminated when the specified timeout expires.

The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock. If the Timers option is not supported, the timeout shall be based on the system clock as returned by the *time*( ) function. The resolution of the timeout shall be the resolution of the clock on which it is based. The **timespec** data type is defined as a structure in the **<time.h>** header.

Under no circumstance shall the function fail with a timeout if the semaphore can be locked immediately. The validity of the *abs_timeout* need not be checked if the semaphore can be locked immediately.

RETURN VALUE

The *sem_timedwait*( ) function shall return zero if the calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore shall be unchanged, and the function shall return a value of –1 and set *errno* to indicate the error.

ERRORS

The *sem_timedwait*( ) function shall fail if:

[EINVAL] The *sem* argument does not refer to a valid semaphore.

[EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1 000 million.

[ETIMEDOUT] The semaphore could not be locked before the specified timeout expired.

The *sem_timedwait*( ) function may fail if:

[EDEADLK] A deadlock condition was detected.

[EINTR] A signal interrupted this function.

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

The *sem_timedwait*( ) function is part of the Semaphores and Timeouts options and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sem_post( ), *sem_trywait*( ), *sem_wait*( ), *semctl*( ), *semget*( ), *semop*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

39342 **NAME**

39343 sem_trywait, sem_wait — lock a semaphore (**REALTIME**)

39344 **SYNOPSIS**

39345 SEM
```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```
39348

39349 **DESCRIPTION**

39350 The *sem_trywait*( ) function shall lock the semaphore referenced by *sem* only if the semaphore is
39351 currently not locked; that is, if the semaphore value is currently positive. Otherwise, shall does
39352 not lock the semaphore.

39353 The *sem_wait*( ) function shall lock the semaphore referenced by *sem* by performing a semaphore
39354 lock operation on that semaphore. If the semaphore value is currently zero, then the calling
39355 thread shall not return from the call to *sem_wait*( ) until it either locks the semaphore or the call is
39356 interrupted by a signal.

39357 Upon successful return, the state of the semaphore shall be locked and shall remain locked until
39358 the *sem_post*( ) function is executed and returns successfully.

39359 The *sem_wait*( ) function is interruptible by the delivery of a signal.

39360 **RETURN VALUE**

39361 The *sem_trywait*( ) and *sem_wait*( ) functions shall return zero if the calling process successfully
39362 performed the semaphore lock operation on the semaphore designated by *sem*. If the call was
39363 unsuccessful, the state of the semaphore shall be unchanged, and the function shall return a
39364 value of –1 and set *errno* to indicate the error.

39365 **ERRORS**

39366 The *sem_trywait*( ) and *sem_wait*( ) functions shall fail if:

39367 [EAGAIN] The semaphore was already locked, so it cannot be immediately locked by the
39368 *sem_trywait*( ) operation (*sem_trywait*( ) only).

39369 [EINVAL] The *sem* argument does not refer to a valid semaphore.

39370 The *sem_trywait*( ) and *sem_wait*( ) functions may fail if:

39371 [EDEADLK] A deadlock condition was detected.

39372 [EINTR] A signal interrupted this function.

39373 **EXAMPLES**

39374 None.

39375 **APPLICATION USAGE**

39376 Applications using these functions may be subject to priority inversion, as discussed in the Base
39377 Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

39378 The *sem_trywait*( ) and *sem_wait*( ) functions are part of the Semaphores option and need not be
39379 provided on all implementations.

39380 **RATIONALE**

39381 None.

FUTURE DIRECTIONS

None.

SEE ALSO

semctl( ), *semget*( ), *semop*( ), *sem_post*( ), *sem_timedwait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sem_trywait*( ) and *sem_wait*( ) functions are marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

NAME

sem_unlink — remove a named semaphore (**REALTIME**)

SYNOPSIS

SEM
```
#include <semaphore.h>

int sem_unlink(const char *name);
```

DESCRIPTION

The *sem_unlink*( ) function shall remove the semaphore named by the string *name*. If the semaphore named by *name* is currently referenced by other processes, then *sem_unlink*( ) shall have no effect on the state of the semaphore. If one or more processes have the semaphore open when *sem_unlink*( ) is called, destruction of the semaphore is postponed until all references to the semaphore have been destroyed by calls to *sem_close*( ), *_exit*( ), or *exec*. Calls to *sem_open*( ) to recreate or reconnect to the semaphore refer to a new semaphore after *sem_unlink*( ) is called. The *sem_unlink*( ) call shall not block until all references have been destroyed; it shall return immediately.

RETURN VALUE

Upon successful completion, the *sem_unlink*( ) function shall return a value of 0. Otherwise, the semaphore shall not be changed and the function shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *sem_unlink*( ) function shall fail if:

[EACCES] Permission is denied to unlink the named semaphore.

[ENAMETOOLONG]
The length of the *name* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

[ENOENT] The named semaphore does not exist.

EXAMPLES

None.

APPLICATION USAGE

The *sem_unlink*( ) function is part of the Semaphores option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

semctl( ), *semget*( ), *semop*( ), *sem_close*( ), *sem_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sem_unlink*( ) function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

39439 **NAME**

39440 sem_wait — lock a semaphore (**REALTIME**)

39441 **SYNOPSIS**

39442 SEM
```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```
39443

39444

39445 **DESCRIPTION**

39446 Refer to *sem_trywait*( ).

NAME

semctl — XSI semaphore control operations

SYNOPSIS

XSI
```
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

DESCRIPTION

The *semctl*( ) function operates on XSI semaphores (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.15, Semaphore). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *semctl*( ) function provides a variety of semaphore control operations as specified by *cmd*. The fourth argument is optional and depends upon the operation requested. If required, it is of type **union semun**, which the application shall explicitly declare:

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short  *array;
} arg;
```

The following semaphore control operations as specified by *cmd* are executed with respect to the semaphore specified by *semid* and *semnum*. The level of permission required for each operation is shown with each command; see Section 2.7 (on page 489). The symbolic names for the values of *cmd* are defined in the **<sys/sem.h>** header:

GETVAL — Return the value of *semval*; see **<sys/sem.h>**. Requires read permission.

SETVAL — Set the value of *semval* to *arg.val*, where *arg* is the value of the fourth argument to *semctl*( ). When this command is successfully executed, the *semadj* value corresponding to the specified semaphore in all processes is cleared. Requires alter permission; see Section 2.7 (on page 489).

GETPID — Return the value of *sempid*. Requires read permission.

GETNCNT — Return the value of *semncnt*. Requires read permission.

GETZCNT — Return the value of *semzcnt*. Requires read permission.

The following values of *cmd* operate on each *semval* in the set of semaphores:

GETALL — Return the value of *semval* for each semaphore in the semaphore set and place into the array pointed to by *arg.array*, where *arg* is the fourth argument to *semctl*( ). Requires read permission.

SETALL — Set the value of *semval* for each semaphore in the semaphore set according to the array pointed to by *arg.array*, where *arg* is the fourth argument to *semctl*( ). When this command is successfully executed, the *semadj* values corresponding to each specified semaphore in all processes are cleared. Requires alter permission.

The following values of *cmd* are also available:

IPC_STAT — Place the current value of each member of the **semid_ds** data structure associated with *semid* into the structure pointed to by *arg.buf*, where *arg* is the fourth argument to *semctl*( ). The contents of this structure are defined in

39491 **<sys/sem.h>**. Requires read permission.

39492 IPC_SET Set the value of the following members of the **semid_ds** data structure
39493 associated with *semid* to the corresponding value found in the structure
39494 pointed to by *arg.buf*, where *arg* is the fourth argument to *semctl*( ):

39495 `sem_perm.uid`
39496 `sem_perm.gid`
39497 `sem_perm.mode`

39498 The mode bits specified in Section 2.7.1 (on page 490) are copied into the
39499 corresponding bits of the *sem_perm.mode* associated with *semid*. The stored
39500 values of any other bits are unspecified.

39501 This command can only be executed by a process that has an effective user ID
39502 equal to either that of a process with appropriate privileges or to the value of
39503 *sem_perm.cuid* or *sem_perm.uid* in the **semid_ds** data structure associated with
39504 *semid*.

39505 IPC_RMID Remove the semaphore identifier specified by *semid* from the system and
39506 destroy the set of semaphores and **semid_ds** data structure associated with it.
39507 This command can only be executed by a process that has an effective user ID
39508 equal to either that of a process with appropriate privileges or to the value of
39509 *sem_perm.cuid* or *sem_perm.uid* in the **semid_ds** data structure associated with
39510 *semid*.

39511 **RETURN VALUE**

39512 If successful, the value returned by *semctl*( ) depends on *cmd* as follows:

39513 GETVAL The value of *semval*.

39514 GETPID The value of *sempid*.

39515 GETNCNT The value of *semncnt*.

39516 GETZCNT The value of *semzcnt*.

39517 All others 0.

39518 Otherwise, *semctl*( ) shall return –1 and set *errno* to indicate the error.

39519 **ERRORS**

39520 The *semctl*( ) function shall fail if:

39521 [EACCES] Operation permission is denied to the calling process; see Section 2.7 (on page
39522 489).

39523 [EINVAL] The value of *semid* is not a valid semaphore identifier, or the value of *semnum*
39524 is less than 0 or greater than or equal to *sem_nsems*, or the value of *cmd* is not a
39525 valid command.

39526 [EPERM] The argument *cmd* is equal to IPC_RMID or IPC_SET and the effective user ID
39527 of the calling process is not equal to that of a process with appropriate
39528 privileges and it is not equal to the value of *sem_perm.cuid* or *sem_perm.uid* in
39529 the data structure associated with *semid*.

39530 [ERANGE] The argument *cmd* is equal to SETVAL or SETALL and the value to which
39531 *semval* is to be set is greater than the system-imposed maximum.

EXAMPLES

None.

APPLICATION USAGE

The fourth parameter in the SYNOPSIS section is now specified as "..." in order to avoid a clash with the ISO C standard when referring to the union *semun* (as defined in Issue 3) and for backward compatibility.

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

semget( ), *semop*( ), *sem_close*( ), *sem_destroy*( ), *sem_getvalue*( ), *sem_init*( ), *sem_open*( ), *sem_post*( ), *sem_unlink*( ), *sem_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/sem.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to the APPLICATION USAGE section.

NAME

semget — get set of XSI semaphores

SYNOPSIS

XSI
```
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

DESCRIPTION

The *semget*( ) function operates on XSI semaphores (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.15, Semaphore). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *semget*( ) function shall return the semaphore identifier associated with *key*.

A semaphore identifier with its associated **semid_ds** data structure and its associated set of *nsems* semaphores (see **<sys/sem.h>**) is created for *key* if one of the following is true:

- The argument *key* is equal to IPC_PRIVATE.
- The argument *key* does not already have a semaphore identifier associated with it and (*semflg* &IPC_CREAT) is non-zero.

Upon creation, the **semid_ds** data structure associated with the new semaphore identifier is initialized as follows:

- In the operation permissions structure *sem_perm.cuid*, *sem_perm.uid*, *sem_perm.cgid*, and *sem_perm.gid* shall be set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of *sem_perm.mode* shall be set equal to the low-order 9 bits of *semflg*.
- The variable *sem_nsems* shall be set equal to the value of *nsems*.
- The variable *sem_otime* shall be set equal to 0 and *sem_ctime* shall be set equal to the current time.
- The data structure associated with each semaphore in the set shall not be initialized. The *semctl*( ) function with the command SETVAL or SETALL can be used to initialize each semaphore.

RETURN VALUE

Upon successful completion, *semget*( ) shall return a non-negative integer, namely a semaphore identifier; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *semget*( ) function shall fail if:

[EACCES] A semaphore identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *semflg* would not be granted; see Section 2.7 (on page 489).

[EEXIST] A semaphore identifier exists for the argument *key* but ((*semflg* &IPC_CREAT) &&(*semflg* &IPC_EXCL)) is non-zero.

[EINVAL] The value of *nsems* is either less than or equal to 0 or greater than the system-imposed limit, or a semaphore identifier exists for the argument *key*, but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to 0.

[ENOENT] A semaphore identifier does not exist for the argument *key* and (*semflg* &IPC_CREAT) is equal to 0.

[ENOSPC] A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system-wide would be exceeded.

EXAMPLES

Creating a Semaphore Identifier

The following example gets a unique semaphore key using the *ftok*() function, then gets a semaphore ID associated with that key using the *semget*() function (the first call also tests to make sure the semaphore exists). If the semaphore does not exist, the program creates it, as shown by the second call to *semget*(). In creating the semaphore for the queuing process, the program attempts to create one semaphore with read/write permission for all. It also uses the IPC_EXCL flag, which forces *semget*() to fail if the semaphore already exists.

After creating the semaphore, the program uses a call to *semop*() to initialize it to the values in the *sbuf* array. The number of processes that can execute concurrently without queuing is initially set to 2. The final call to *semget*() creates a semaphore identifier that can be used later in the program.

```
#include <sys/types.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <pwd.h>
#include <fcntl.h>
#include <limits.h>
...
key_t semkey;
int semid, pfd, fv;
struct sembuf sbuf;
char *lgn;
char filename[PATH_MAX+1];
struct stat outstat;
struct passwd *pw;
...
/* Get unique key for semaphore. */
if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
    perror("IPC error: ftok"); exit(1);
}

/* Get semaphore ID associated with this key. */
if ((semid = semget(semkey, 0, 0)) == -1) {

    /* Semaphore does not exist - Create. */
    if ((semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR |
        S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) != -1)
    {
        /* Initialize the semaphore. */
        sbuf.sem_num = 0;
```

```
                sbuf.sem_op = 2;  /* This is the number of runs without queuing. */
                sbuf.sem_flg = 0;
                if (semop(semid, &sbuf, 1) == -1) {
                    perror("IPC error: semop"); exit(1);
                }
            }
            else if (errno == EEXIST) {
                if ((semid = semget(semkey, 0, 0)) == -1) {
                    perror("IPC error 1: semget"); exit(1);
                }
            }
            else {
                perror("IPC error 2: semget"); exit(1);
            }
        }
        ...
```

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

semctl( ), *semop*( ), *sem_close*( ), *sem_destroy*( ), *sem_getvalue*( ), *sem_init*( ), *sem_open*( ), *sem_post*( ), *sem_unlink*( ), *sem_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/sem.h>**, Section 2.7 (on page 489).

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

NAME

semop — XSI semaphore operations

SYNOPSIS

XSI
```
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
```

DESCRIPTION

The *semop*( ) function operates on XSI semaphores (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.15, Semaphore). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *semop*( ) function shall perform atomically a user-defined array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by the argument *semid*.

The argument *sops* is a pointer to a user-defined array of semaphore operation structures. The implementation shall not modify elements of this array unless the application uses implementation-defined extensions.

The argument *nsops* is the number of such structures in the array.

Each structure, **sembuf**, includes the following members:

Member Type	Member Name	Description
short	*sem_num*	Semaphore number.
short	*sem_op*	Semaphore operation.
short	*sem_flg*	Operation flags.

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*.

The variable *sem_op* specifies one of three semaphore operations:

1. If *sem_op* is a negative integer and the calling process has alter permission, one of the following shall occur:
   - If *semval*(see **<sys/sem.h>**) is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from *semval*. Also, if (*sem_flg* &SEM_UNDO) is non-zero, the absolute value of *sem_op* shall be added to the calling process' *semadj* value for the specified semaphore.
   - If *semval* is less than the absolute value of *sem_op* and (*sem_flg* &IPC_NOWAIT) is non-zero, *semop*( ) shall return immediately.
   - If *semval* is less than the absolute value of *sem_op* and (*sem_flg* &IPC_NOWAIT) is 0, *semop*( ) shall increment the *semncnt* associated with the specified semaphore and suspend execution of the calling thread until one of the following conditions occurs:
     — The value of *semval* becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of *semncnt* associated with the specified semaphore shall be decremented, the absolute value of *sem_op* shall be subtracted from *semval* and, if (*sem_flg* &SEM_UNDO) is non-zero, the absolute value of *sem_op* shall be added to the calling process' *semadj* value for the specified semaphore.
     — The *semid* for which the calling thread is awaiting action is removed from the system. When this occurs, *errno* shall be set equal to [EIDRM] and –1 shall be

returned.

— The calling thread receives a signal that is to be caught. When this occurs, the value of *semncnt* associated with the specified semaphore shall be decremented, and the calling thread shall resume execution in the manner prescribed in *sigaction*( ).

2. If *sem_op* is a positive integer and the calling process has alter permission, the value of *sem_op* shall be added to *semval* and, if (*sem_flg* &SEM_UNDO) is non-zero, the value of *sem_op* shall be subtracted from the calling process' *semadj* value for the specified semaphore.

3. If *sem_op* is 0 and the calling process has read permission, one of the following shall occur:

   - If *semval* is 0, *semop*( ) shall return immediately.
   - If *semval* is non-zero and (*sem_flg* &IPC_NOWAIT) is non-zero, *semop*( ) shall return immediately.
   - If *semval* is non-zero and (*sem_flg* &IPC_NOWAIT) is 0, *semop*( ) shall increment the *semzcnt* associated with the specified semaphore and suspend execution of the calling thread until one of the following occurs:

     — The value of *semval* becomes 0, at which time the value of *semzcnt* associated with the specified semaphore shall be decremented.

     — The *semid* for which the calling thread is awaiting action is removed from the system. When this occurs, *errno* shall be set equal to [EIDRM] and –1 shall be returned.

     — The calling thread receives a signal that is to be caught. When this occurs, the value of *semzcnt* associated with the specified semaphore shall be decremented, and the calling thread shall resume execution in the manner prescribed in *sigaction*( ).

Upon successful completion, the value of *sempid* for each semaphore specified in the array pointed to by *sops* shall be set equal to the process ID of the calling process.

RETURN VALUE

Upon successful completion, *semop*( ) shall return 0; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *semop*( ) function shall fail if:

[E2BIG]	The value of *nsops* is greater than the system-imposed maximum.
[EACCES]	Operation permission is denied to the calling process; see Section 2.7 (on page 489).
[EAGAIN]	The operation would result in suspension of the calling process but (*sem_flg* &IPC_NOWAIT) is non-zero.
[EFBIG]	The value of *sem_num* is less than 0 or greater than or equal to the number of semaphores in the set associated with *semid*.
[EIDRM]	The semaphore identifier *semid* is removed from the system.
[EINTR]	The *semop*( ) function was interrupted by a signal.
[EINVAL]	The value of *semid* is not a valid semaphore identifier, or the number of individual semaphores for which the calling process requests a SEM_UNDO would exceed the system-imposed limit.

[ENOSPC] The limit on the number of individual processes requesting a SEM_UNDO would be exceeded.

[ERANGE] An operation would cause a *semval* to overflow the system-imposed limit, or an operation would cause a *semadj* value to overflow the system-imposed limit.

EXAMPLES

Setting Values in Semaphores

The following example sets the values of the two semaphores associated with the *semid* identifier to the values contained in the *sb* array.

```
#include <sys/sem.h>
...
int semid;
struct sembuf sb[2];
int nsops = 2;
int result;

/* Adjust value of semaphore in the semaphore array semid. */
sb[0].sem_num = 0;
sb[0].sem_op = -1;
sb[0].sem_flg = SEM_UNDO | IPC_NOWAIT;
sb[1].sem_num = 1;
sb[1].sem_op =  1;
sb[1].sem_flg = 0;

result = semop(semid, sb, nsops);
```

Creating a Semaphore Identifier

The following example gets a unique semaphore key using the *ftok*() function, then gets a semaphore ID associated with that key using the *semget*() function (the first call also tests to make sure the semaphore exists). If the semaphore does not exist, the program creates it, as shown by the second call to *semget*(). In creating the semaphore for the queuing process, the program attempts to create one semaphore with read/write permission for all. It also uses the IPC_EXCL flag, which forces *semget*() to fail if the semaphore already exists.

After creating the semaphore, the program uses a call to *semop*() to initialize it to the values in the *sbuf* array. The number of processes that can execute concurrently without queuing is initially set to 2. The final call to *semget*() creates a semaphore identifier that can be used later in the program.

The final call to *semop*() acquires the semaphore and waits until it is free; the SEM_UNDO option releases the semaphore when the process exits, waiting until there are less than two processes running concurrently.

```
#include <sys/types.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
```

```
#include <pwd.h>
#include <fcntl.h>
#include <limits.h>
...
key_t semkey;
int semid, pfd, fv;
struct sembuf sbuf;
char *lgn;
char filename[PATH_MAX+1];
struct stat outstat;
struct passwd *pw;
...
/* Get unique key for semaphore. */
if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
    perror("IPC error: ftok"); exit(1);
}

/* Get semaphore ID associated with this key. */
if ((semid = semget(semkey, 0, 0)) == -1) {

    /* Semaphore does not exist - Create. */
    if ((semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR |
        S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) != -1)
    {
        /* Initialize the semaphore. */
        sbuf.sem_num = 0;
        sbuf.sem_op = 2;  /* This is the number of runs without queuing. */
        sbuf.sem_flg = 0;
        if (semop(semid, &sbuf, 1) == -1) {
            perror("IPC error: semop"); exit(1);
        }
    }
    else if (errno == EEXIST) {
        if ((semid = semget(semkey, 0, 0)) == -1) {
            perror("IPC error 1: semget"); exit(1);
        }
    }
    else {
        perror("IPC error 2: semget"); exit(1);
    }
}
...
sbuf.sem_num = 0;
sbuf.sem_op = -1;
sbuf.sem_flg = SEM_UNDO;
if (semop(semid, &sbuf, 1) == -1) {
    perror("IPC Error: semop"); exit(1);
}
```

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit*( ), *fork*( ), *semctl*( ), *semget*( ), *sem_close*( ), *sem_destroy*( ), *sem_getvalue*( ), *sem_init*( ), *sem_open*( ), *sem_post*( ), *sem_unlink*( ), *sem_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/ipc.h>**, **<sys/sem.h>**, **<sys/types.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

39875 **NAME**

39876 send — send a message on a socket

39877 **SYNOPSIS**

39878 `#include <sys/socket.h>`

39879 `ssize_t send(int socket, const void *buffer, size_t length, int flags);`

39880 **DESCRIPTION**

39881 The *send*( ) function shall initiate transmission of a message from the specified socket to its peer.
39882 The *send*( ) function shall send a message only when the socket is connected (including when the
39883 peer of a connectionless socket has been set via *connect*( )).

39884 The *send*( ) functions takes the following arguments:

39885 *socket* Specifies the socket file descriptor.

39886 *buffer* Points to the buffer containing the message to send.

39887 *length* Specifies the length of the message in bytes.

39888 *flags* Specifies the type of message transmission. Values of this argument are
39889 formed by logically OR'ing zero or more of the following flags:

39890 MSG_EOR Terminates a record (if supported by the protocol).

39891 MSG_OOB Sends out-of-band data on sockets that support out-of-band
39892 communications. The significance and semantics of out-of-
39893 band data are protocol-specific.

39894 The length of the message to be sent is specified by the *length* argument. If the message is too
39895 long to pass through the underlying protocol, *send*( ) shall fail and no data shall be transmitted.

39896 Successful completion of a call to *send*( ) does not guarantee delivery of the message. A return
39897 value of –1 indicates only locally-detected errors.

39898 If space is not available at the sending socket to hold the message to be transmitted, and the
39899 socket file descriptor does not have O_NONBLOCK set, *send*( ) shall block until space is
39900 available. If space is not available at the sending socket to hold the message to be transmitted,
39901 and the socket file descriptor does have O_NONBLOCK set, *send*( ) shall fail. The *select*( ) and
39902 *poll*( ) functions can be used to determine when it is possible to send more data.

39903 The socket in use may require the process to have appropriate privileges to use the *send*( )
39904 function.

39905 **RETURN VALUE**

39906 Upon successful completion, *send*( ) shall return the number of bytes sent. Otherwise, –1 shall be
39907 returned and *errno* set to indicate the error.

39908 **ERRORS**

39909 The *send*( ) function shall fail if:

39910 [EAGAIN] or [EWOULDBLOCK]
39911 The socket's file descriptor is marked O_NONBLOCK and the requested
39912 operation would block.

39913 [EBADF] The *socket* argument is not a valid file descriptor.

39914 [ECONNRESET] A connection was forcibly closed by a peer.

39915 [EDESTADDRREQ]
39916 The socket is not connection-mode and no peer address is set.

39917 [EINTR] A signal interrupted *send*( ) before any data was transmitted.

39918 [EMSGSIZE] The message is too large be sent all at once, as the socket requires.

39919 [ENOTCONN] The socket is not connected or otherwise has not had the peer pre-specified.

39920 [ENOTSOCK] The *socket* argument does not refer to a socket.

39921 [EOPNOTSUPP] The *socket* argument is associated with a socket that does not support one or
39922 more of the values set in *flags*.

39923 [EPIPE] The socket is shut down for writing, or the socket is connection-mode and is
39924 no longer connected. In the latter case, and if the socket is of type
39925 SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.

39926 The *send*( ) function may fail if:

39927 [EACCES] The calling process does not have the appropriate privileges.

39928 [EIO] An I/O error occurred while reading from or writing to the file system.

39929 [ENETDOWN] The local network interface used to reach the destination is down.

39930 [ENETUNREACH]
39931 No route to the network is present.

39932 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

39933 **EXAMPLES**

39934 None.

39935 **APPLICATION USAGE**

39936 The *send*( ) function is equivalent to *sendto*( ) with a null pointer *dest_len* argument, and to *write*( ) |
39937 if no flags are used.

39938 **RATIONALE**

39939 None.

39940 **FUTURE DIRECTIONS**

39941 None.

39942 **SEE ALSO**

39943 *connect*( ), *getsockopt*( ), *poll*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *sendmsg*( ), *sendto*( ),
39944 *setsockopt*( ), *shutdown*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
39945 **<sys/socket.h>**

39946 **CHANGE HISTORY**

39947 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

sendmsg — send a message on a socket using a message structure

SYNOPSIS

```
#include <sys/socket.h>

ssize_t sendmsg(int socket, const struct msghdr *message, int flags);
```

DESCRIPTION

The *sendmsg*( ) function shall send a message through a connection-mode or connectionless-mode socket. If the socket is connectionless-mode, the message shall be sent to the address specified by **msghdr**. If the socket is connection-mode, the destination address in **msghdr** shall be ignored.

The *sendmsg*( ) function takes the following arguments:

socket Specifies the socket file descriptor.

message Points to a **msghdr** structure, containing both the destination address and the buffers for the outgoing message. The length and format of the address depend on the address family of the socket. The *msg_flags* member is ignored.

flags Specifies the type of message transmission. The application may specify 0 or the following flag:

MSG_EOR Terminates a record (if supported by the protocol).

MSG_OOB Sends out-of-band data on sockets that support out-of-bound data. The significance and semantics of out-of-band data are protocol-specific.

The *msg_iov* and *msg_iovlen* fields of *message* specify zero or more buffers containing the data to be sent. *msg_iov* points to an array of **iovec** structures; *msg_iovlen* shall be set to the dimension of this array. In each **iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field gives its size in bytes. Some of these sizes can be zero. The data from each storage area indicated by *msg_iov* is sent in turn.

Successful completion of a call to *sendmsg*( ) does not guarantee delivery of the message. A return value of –1 indicates only locally-detected errors.

If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does not have O_NONBLOCK set, *sendmsg*( ) function shall block until space is available. If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does have O_NONBLOCK set, *sendmsg*( ) function shall fail.

If the socket protocol supports broadcast and the specified address is a broadcast address for the socket protocol, *sendmsg*( ) shall fail if the SO_BROADCAST option is not set for the socket.

The socket in use may require the process to have appropriate privileges to use the *sendmsg*( ) function.

RETURN VALUE

Upon successful completion, *sendmsg*( ) shall return the number of bytes sent. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *sendmsg*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]
The socket's file descriptor is marked O_NONBLOCK and the requested

operation would block.

[EAFNOSUPPORT]
Addresses in the specified address family cannot be used with this socket.

[EBADF] The *socket* argument is not a valid file descriptor.

[ECONNRESET] A connection was forcibly closed by a peer.

[EINTR] A signal interrupted *sendmsg*( ) before any data was transmitted.

[EINVAL] The sum of the *iov_len* values overflows an **ssize_t**.

[EMSGSIZE] The message is too large to be sent all at once (as the socket requires), or the *msg_iovlen* member of the **msghdr** structure pointed to by *message* is less than or equal to 0 or is greater than {IOV_MAX}.

[ENOTCONN] The socket is connection-mode but is not connected.

[ENOTSOCK] The *socket* argument does not refer a socket.

[EOPNOTSUPP] The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.

[EPIPE] The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.

If the address family of the socket is AF_UNIX, then *sendmsg*( ) shall fail if:

[EIO] An I/O error occurred while reading from or writing to the file system.

[ELOOP] A loop exists in symbolic links encountered during resolution of the pathname in the socket address.

[ENAMETOOLONG]
A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.

[ENOENT] A component of the pathname does not name an existing file or the path name is an empty string.

[ENOTDIR] A component of the path prefix of the pathname in the socket address is not a directory.

The *sendmsg*( ) function may fail if:

[EACCES] Search permission is denied for a component of the path prefix; or write access to the named socket is denied.

[EDESTADDRREQ]
The socket is not connection-mode and does not have its peer address set, and no destination address was specified.

[EHOSTUNREACH]
The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).

[EIO] An I/O error occurred while reading from or writing to the file system.

[EISCONN] A destination address was specified and the socket is already connected.

[ENETDOWN] The local network interface used to reach the destination is down.

[ENETUNREACH]
No route to the network is present.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

If the address family of the socket is AF_UNIX, then *sendmsg*( ) may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in the socket address. |

[ENAMETOOLONG]
Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}. |

EXAMPLES

Done.

APPLICATION USAGE

The *select*( ) and *poll*( ) functions can be used to determine when it is possible to send more data.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getsockopt( ), *poll*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *send*( ), *sendto*( ), *setsockopt*( ), *shutdown*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

40057 **NAME**

40058 sendto — send a message on a socket

40059 **SYNOPSIS**

40060
```
#include <sys/socket.h>
```

40061
```
ssize_t sendto(int socket, const void *message, size_t length,
    int flags, const struct sockaddr *dest_addr,
    socklen_t dest_len);
```

40064 **DESCRIPTION**

40065 The *sendto*( ) function shall send a message through a connection-mode or connectionless-mode
40066 socket. If the socket is connectionless-mode, the message shall be sent to the address specified by
40067 *dest_addr*. If the socket is connection-mode, *dest_addr* shall be ignored. |

40068 The *sendto*( ) function takes the following arguments:

40069 *socket* Specifies the socket file descriptor.

40070 *message* Points to a buffer containing the message to be sent.

40071 *length* Specifies the size of the message in bytes.

40072 *flags* Specifies the type of message transmission. Values of this argument are
40073 formed by logically OR'ing zero or more of the following flags:

40074 MSG_EOR Terminates a record (if supported by the protocol).

40075 MSG_OOB Sends out-of-band data on sockets that support out-of-band
40076 data. The significance and semantics of out-of-band data are
40077 protocol-specific.

40078 *dest_addr* Points to a **sockaddr** structure containing the destination address. The length
40079 and format of the address depend on the address family of the socket.

40080 *dest_len* Specifies the length of the **sockaddr** structure pointed to by the *dest_addr*
40081 argument.

40082 If the socket protocol supports broadcast and the specified address is a broadcast address for the
40083 socket protocol, *sendto*( ) shall fail if the SO_BROADCAST option is not set for the socket.

40084 The *dest_addr* argument specifies the address of the target. The *length* argument specifies the
40085 length of the message.

40086 Successful completion of a call to *sendto*( ) does not guarantee delivery of the message. A return
40087 value of –1 indicates only locally-detected errors.

40088 If space is not available at the sending socket to hold the message to be transmitted and the
40089 socket file descriptor does not have O_NONBLOCK set, *sendto*( ) shall block until space is |
40090 available. If space is not available at the sending socket to hold the message to be transmitted |
40091 and the socket file descriptor does have O_NONBLOCK set, *sendto*( ) shall fail.

40092 The socket in use may require the process to have appropriate privileges to use the *sendto*( )
40093 function.

40094 **RETURN VALUE**

40095 Upon successful completion, *sendto*( ) shall return the number of bytes sent. Otherwise, –1 shall
40096 be returned and *errno* set to indicate the error.

40097 **ERRORS**

40098 The *sendto*( ) function shall fail if:

40099 [EAFNOSUPPORT]
40100 Addresses in the specified address family cannot be used with this socket.

40101 [EAGAIN] or [EWOULDBLOCK]
40102 The socket's file descriptor is marked O_NONBLOCK and the requested
40103 operation would block.

40104 [EBADF] The *socket* argument is not a valid file descriptor.

40105 [ECONNRESET] A connection was forcibly closed by a peer.

40106 [EINTR] A signal interrupted *sendto*( ) before any data was transmitted.

40107 [EMSGSIZE] The message is too large to be sent all at once, as the socket requires.

40108 [ENOTCONN] The socket is connection-mode but is not connected.

40109 [ENOTSOCK] The *socket* argument does not refer to a socket.

40110 [EOPNOTSUPP] The *socket* argument is associated with a socket that does not support one or
40111 more of the values set in *flags*.

40112 [EPIPE] The socket is shut down for writing, or the socket is connection-mode and is
40113 no longer connected. In the latter case, and if the socket is of type
40114 SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.

40115 If the address family of the socket is AF_UNIX, then *sendto*( ) shall fail if:

40116 [EIO] An I/O error occurred while reading from or writing to the file system.

40117 [ELOOP] A loop exists in symbolic links encountered during resolution of the pathname |
40118 in the socket address. |

40119 [ENAMETOOLONG]
40120 A component of a pathname exceeded {NAME_MAX} characters, or an entire |
40121 pathname exceeded {PATH_MAX} characters. |

40122 [ENOENT] A component of the pathname does not name an existing file or the pathname |
40123 is an empty string. |

40124 [ENOTDIR] A component of the path prefix of the pathname in the socket address is not a |
40125 directory.

40126 The *sendto*( ) function may fail if:

40127 [EACCES] Search permission is denied for a component of the path prefix; or write
40128 access to the named socket is denied.

40129 [EDESTADDRREQ]
40130 The socket is not connection-mode and does not have its peer address set, and
40131 no destination address was specified.

40132 [EHOSTUNREACH]
40133 The destination host cannot be reached (probably because the host is down or
40134 a remote router cannot reach it).

40135 [EINVAL] The *dest_len* argument is not a valid length for the address family.

40136 [EIO] An I/O error occurred while reading from or writing to the file system.

[EISCONN] A destination address was specified and the socket is already connected. This error may or may not be returned for connection mode sockets.

[ENETDOWN] The local network interface used to reach the destination is down.

[ENETUNREACH] No route to the network is present.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

If the address family of the socket is AF_UNIX, then *sendto*( ) may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in the socket address. |

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}. |

EXAMPLES

None.

APPLICATION USAGE

The *select*( ) and *poll*( ) functions can be used to determine when it is possible to send more data.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getsockopt( ), *poll*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *send*( ), *sendmsg*( ), *setsockopt*( ), *shutdown*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

40165 **NAME**

40166 setbuf — assign buffering to a stream

40167 **SYNOPSIS**

40168
```
#include <stdio.h>
```

40169
```
void setbuf(FILE *restrict stream, char *restrict buf);
```

40170 **DESCRIPTION**

40171 CX The functionality described on this reference page is aligned with the ISO C standard. Any
40172 conflict between the requirements described here and the ISO C standard is unintentional. This
40173 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

40174 Except that it returns no value, the function call:

40175
```
setbuf(stream, buf)
```

40176 shall be equivalent to:

40177
```
setvbuf(stream, buf, _IOFBF, BUFSIZ)
```

40178 if *buf* is not a null pointer, or to:

40179
```
setvbuf(stream, buf, _IONBF, BUFSIZ)
```

40180 if *buf* is a null pointer.

40181 **RETURN VALUE**

40182 The *setbuf*( ) function shall not return a value.

40183 **ERRORS**

40184 No errors are defined.

40185 **EXAMPLES**

40186 None.

40187 **APPLICATION USAGE**

40188 A common source of error is allocating buffer space as an ‘‘automatic’’ variable in a code block,
40189 and then failing to close the stream in the same block.

40190 With *setbuf*( ), allocating a buffer of BUFSIZ bytes does not necessarily imply that all of BUFSIZ
40191 bytes are used for the buffer area.

40192 **RATIONALE**

40193 None.

40194 **FUTURE DIRECTIONS**

40195 None.

40196 **SEE ALSO**

40197 *fopen*( ), *setvbuf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

40198 **CHANGE HISTORY**

40199 First released in Issue 1. Derived from Issue 1 of the SVID.

40200 **Issue 6**

40201 The prototype for *setbuf*( ) is updated for alignment with the ISO / IEC 9899: 1999 standard.

40202 **NAME**

40203 setcontext — set current user context

40204 **SYNOPSIS**

40205 XSI
```
#include <ucontext.h>
```

40206
```
int setcontext(const ucontext_t *ucp);
```

40207

40208 **DESCRIPTION**

40209 Refer to *getcontext*( ).

40210 **NAME**

40211 setegid — set effective group ID

40212 **SYNOPSIS**

40213 `#include <unistd.h>`

40214 `int setegid(gid_t gid);`

40215 **DESCRIPTION**

40216 If *gid* is equal to the real group ID or the saved set-group-ID, or if the process has appropriate
40217 privileges, *setegid*( ) shall set the effective group ID of the calling process to *gid*; the real group
40218 ID, saved set-group-ID, and any supplementary group IDs shall remain unchanged.

40219 The *setegid*( ) function shall not affect the supplementary group list in any way.

40220 **RETURN VALUE**

40221 Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to
40222 indicate the error.

40223 **ERRORS**

40224 The *setegid*( ) function shall fail if:

40225 [EINVAL] The value of the *gid* argument is invalid and is not supported by the
40226 implementation.

40227 [EPERM] The process does not have appropriate privileges and *gid* does not match the
40228 real group ID or the saved set-group-ID.

40229 **EXAMPLES**

40230 None.

40231 **APPLICATION USAGE**

40232 None.

40233 **RATIONALE**

40234 Refer to the RATIONALE section in *setuid*( ).

40235 **FUTURE DIRECTIONS**

40236 None.

40237 **SEE ALSO**

40238 *exec*, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the
40239 Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

40240 **CHANGE HISTORY**

40241 First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

40242 **NAME**

40243 setenv — add or change environment variable

40244 **SYNOPSIS**

40245 CX
```
#include <stdlib.h>

int setenv(const char *envname, const char *envval, int overwrite);
```
40246

40247

40248 **DESCRIPTION**

40249 The *setenv*( ) function shall update or add a variable in the environment of the calling process.
40250 The *envname* argument points to a string containing the name of an environment variable to be
40251 added or altered. The environment variable shall be set to the value to which *envval* points. The
40252 function shall fail if *envname* points to a string which contains an '=' character. If the
40253 environment variable named by *envname* already exists and the value of *overwrite* is non-zero,
40254 the function shall return success and the environment shall be updated. If the environment
40255 variable named by *envname* already exists and the value of *overwrite* is zero, the function shall
40256 return success and the environment shall remain unchanged.

40257 If the application modifies *environ* or the pointers to which it points, the behavior of *setenv*( ) is
40258 undefined. The *setenv*( ) function shall update the list of pointers to which *environ* points.

40259 The strings described by *envname* and *envval* are copied by this function.

40260 The *setenv*( ) function need not be reentrant. A function that is not required to be reentrant is not
40261 required to be thread-safe.

40262 **RETURN VALUE**

40263 Upon successful completion, zero shall be returned. Otherwise, –1 shall be returned, *errno* set to
40264 indicate the error, and the environment shall be unchanged.

40265 **ERRORS**

40266 The *setenv*( ) function shall fail if:

40267 [EINVAL] The *name* argument is a null pointer, points to an empty string, or points to a
40268 string containing an '=' character.

40269 [ENOMEM] Insufficient memory was available to add a variable or its value to the
40270 environment.

40271 **EXAMPLES**

40272 None.

40273 **APPLICATION USAGE**

40274 None.

40275 **RATIONALE**

40276 Unanticipated results may occur if *setenv*( ) changes the external variable *environ*. In particular,
40277 if the optional *envp* argument to *main*( ) is present, it is not changed, and thus may point to an
40278 obsolete copy of the environment (as may any other copy of *environ*). However, other than the
40279 aforementioned restriction, the developers of IEEE Std 1003.1-200x intended that the traditional
40280 method of walking through the environment by way of the *environ* pointer must be supported.

40281 It was decided that *setenv*( ) should be required by this revision because it addresses a piece of
40282 missing functionality, and does not impose a significant burden on the implementor.

40283 There was considerable debate as to whether the System V *putenv*( ) function or the BSD *setenv*( )
40284 function should be required as a mandatory function. The *setenv*( ) function was chosen because
40285 it permitted the implementation of *unsetenv*( ) function to delete environmental variables,
40286 without specifying an additional interface. The *putenv*( ) function is available as an XSI

extension.

The standard developers considered requiring that *setenv*( ) indicate an error when a call to it would result in exceeding {ARG_MAX}. The requirement was rejected since the condition might be temporary, with the application eventually reducing the environment size. The ultimate success or failure depends on the size at the time of a call to *exec*, which returns an indication of this error condition.

FUTURE DIRECTIONS

None.

SEE ALSO

getenv( ), *unsetenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

40300 **NAME**

40301 seteuid — set effective user ID

40302 **SYNOPSIS**

40303
```
#include <unistd.h>
```

40304
```
int seteuid(uid_t uid);
```

40305 **DESCRIPTION**

40306 If *uid* is equal to the real user ID or the saved set-user-ID, or if the process has appropriate
40307 privileges, *seteuid*( ) shall set the effective user ID of the calling process to *uid*; the real user ID
40308 and saved set-user-ID shall remain unchanged.

40309 The *seteuid*( ) function shall not affect the supplementary group list in any way.

40310 **RETURN VALUE**

40311 Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to
40312 indicate the error.

40313 **ERRORS**

40314 The *seteuid*( ) function shall fail if:

40315 [EINVAL] The value of the *uid* argument is invalid and is not supported by the
40316 implementation.

40317 [EPERM] The process does not have appropriate privileges and *uid* does not match the
40318 real group ID or the saved set-group-ID.

40319 **EXAMPLES**

40320 None.

40321 **APPLICATION USAGE**

40322 None.

40323 **RATIONALE**

40324 Refer to the RATIONALE section in *setuid*( ).

40325 **FUTURE DIRECTIONS**

40326 None.

40327 **SEE ALSO**

40328 *exec*, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the
40329 Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

40330 **CHANGE HISTORY**

40331 First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

40332 **NAME**

40333 setgid — set-group-ID

40334 **SYNOPSIS**

40335 `#include <unistd.h>`

40336 `int setgid(gid_t gid);`

40337 **DESCRIPTION**

40338 If the process has appropriate privileges, *setgid*( ) shall set the real group ID, effective group ID,
40339 and the saved set-group-ID of the calling process to *gid*.

40340 If the process does not have appropriate privileges, but *gid* is equal to the real group ID or the
40341 saved set-group-ID, *setgid*( ) shall set the effective group ID to *gid*; the real group ID and saved
40342 set-group-ID shall remain unchanged.

40343 The *setgid*( ) function shall not affect the supplementary group list in any way.

40344 Any supplementary group IDs of the calling process shall remain unchanged.

40345 **RETURN VALUE**

40346 Upon successful completion, 0 is returned. Otherwise, –1 shall be returned and *errno* set to
40347 indicate the error.

40348 **ERRORS**

40349 The *setgid*( ) function shall fail if:

40350 [EINVAL] The value of the *gid* argument is invalid and is not supported by the
40351 implementation.

40352 [EPERM] The process does not have appropriate privileges and *gid* does not match the
40353 real group ID or the saved set-group-ID.

40354 **EXAMPLES**

40355 None.

40356 **APPLICATION USAGE**

40357 None.

40358 **RATIONALE**

40359 Refer to the RATIONALE section in *setuid*( ).

40360 **FUTURE DIRECTIONS**

40361 None.

40362 **SEE ALSO**

40363 *exec*, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the
40364 Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

40365 **CHANGE HISTORY**

40366 First released in Issue 1. Derived from Issue 1 of the SVID.

40367 **Issue 6**

40368 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

40369 The following new requirements on POSIX implementations derive from alignment with the
40370 Single UNIX Specification:

40371 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
40372 required for conforming implementations of previous POSIX specifications, it was not
40373 required for UNIX applications.

- Functionality associated with _POSIX_SAVED_IDS is now mandated. This is a FIPS requirement.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effects of *setgid*( ) in processes without appropriate privileges are changed
- A requirement that the supplementary group list is not affected is added.

40379 **NAME**

40380 setgrent — reset group database to first entry

40381 **SYNOPSIS**

40382 XSI
```
#include <grp.h>

void setgrent(void);
```
40383

40384

40385 **DESCRIPTION**

40386 Refer to *endgrent*( ).

40387 **NAME**

40388 sethostent — network host database functions

40389 **SYNOPSIS**

40390 `#include <netdb.h>`

40391 `void sethostent(int stayopen);`

40392 **DESCRIPTION**

40393 Refer to *endhostent*( ).

40394 **NAME**

40395 setitimer — set value of interval timer

40396 **SYNOPSIS**

40397 XSI
```
#include <sys/time.h>

int setitimer(int which, const struct itimerval *restrict value,
    struct itimerval *restrict ovalue);
```
40400

40401 **DESCRIPTION**

40402 Refer to *getitimer*( ).

40403 **NAME**

40404 setjmp — set jump point for a non-local goto

40405 **SYNOPSIS**

40406
```
#include <setjmp.h>
```

40407
```
int setjmp(jmp_buf env);
```

40408 **DESCRIPTION**

40409 CX The functionality described on this reference page is aligned with the ISO C standard. Any
40410 conflict between the requirements described here and the ISO C standard is unintentional. This
40411 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

40412 A call to *setjmp*( ), shall save the calling environment in its *env* argument for later use by
40413 *longjmp*( ).

40414 It is unspecified whether *setjmp*( ) is a macro or a function. If a macro definition is suppressed in
40415 order to access an actual function, or a program defines an external identifier with the name
40416 *setjmp*, the behavior is undefined.

40417 An application shall ensure that an invocation of *setjmp*( ) appears in one of the following
40418 contexts only:

40419 - The entire controlling expression of a selection or iteration statement

40420 - One operand of a relational or equality operator with the other operand an integral constant
40421 expression, with the resulting expression being the entire controlling expression of a
40422 selection or iteration statement

40423 - The operand of a unary '!' operator with the resulting expression being the entire
40424 controlling expression of a selection or iteration

40425 - The entire expression of an expression statement (possibly cast to **void**)

40426 If the invocation appears in any other context, the behavior is undefined.

40427 **RETURN VALUE**

40428 If the return is from a direct invocation, *setjmp*( ) shall return 0. If the return is from a call to
40429 *longjmp*( ), *setjmp*( ) shall return a non-zero value.

40430 **ERRORS**

40431 No errors are defined.

40432 **EXAMPLES**

40433 None.

40434 **APPLICATION USAGE**

40435 In general, *sigsetjmp*( ) is more useful in dealing with errors and interrupts encountered in a low-
40436 level subroutine of a program.

40437 **RATIONALE**

40438 None.

40439 **FUTURE DIRECTIONS**

40440 None.

40441 **SEE ALSO**

40442 *longjmp*( ), *sigsetjmp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<setjmp.h>**

40443 **CHANGE HISTORY**

40444 First released in Issue 1. Derived from Issue 1 of the SVID.

40445 **Issue 6**

40446 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

40447 **NAME**

40448 setkey — set encoding key (**CRYPT**)

40449 **SYNOPSIS**

40450 XSI
```
#include <stdlib.h>

void setkey(const char *key);
```
40451

40452

40453 **DESCRIPTION**

40454 The *setkey*( ) function provides access to an implementation-defined encoding algorithm. The |
40455 argument of *setkey*( ) is an array of length 64 bytes containing only the bytes with numerical |
40456 value of 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is
40457 ignored; this gives a 56-bit key which is used by the algorithm. This is the key that shall be used
40458 with the algorithm to encode a string *block* passed to *encrypt*( ).

40459 The *setkey*( ) function shall not change the setting of *errno* if successful. An application wishing to
40460 check for error situations should set *errno* to 0 before calling *setkey*( ). If *errno* is non-zero on
40461 return, an error has occurred.

40462 The *setkey*( ) function need not be reentrant. A function that is not required to be reentrant is not
40463 required to be thread-safe.

40464 **RETURN VALUE**

40465 No values are returned.

40466 **ERRORS**

40467 The *setkey*( ) function shall fail if:

40468 [ENOSYS] The functionality is not supported on this implementation.

40469 **EXAMPLES**

40470 None.

40471 **APPLICATION USAGE**

40472 Decoding need not be implemented in all environments. This is related to government |
40473 restrictions in some countries on encryption and decryption routines. Historical practice has |
40474 been to ship a different version of the encryption library without the decryption feature in the |
40475 routines supplied. Thus the exported version of *encrypt*( ) does encoding but not decoding. |

40476 **RATIONALE**

40477 None.

40478 **FUTURE DIRECTIONS**

40479 None.

40480 **SEE ALSO**

40481 *crypt*( ), *encrypt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

40482 **CHANGE HISTORY**

40483 First released in Issue 1. Derived from Issue 1 of the SVID.

40484 **Issue 5**

40485 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

40486 **NAME**

40487 setlocale — set program locale

40488 **SYNOPSIS**

40489 `#include <locale.h>`

40490 `char *setlocale(int` *`category`*`, const char *`*`locale`*`);`

40491 **DESCRIPTION**

40492 CX The functionality described on this reference page is aligned with the ISO C standard. Any
40493 conflict between the requirements described here and the ISO C standard is unintentional. This
40494 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

40495 The *setlocale*( ) function selects the appropriate piece of the program's locale, as specified by the
40496 *category* and *locale* arguments, and may be used to change or query the program's entire locale or
40497 portions thereof. The value *LC_ALL* for *category* names the program's entire locale; other values
40498 for *category* name only a part of the program's locale:

40499 *LC_COLLATE* Affects the behavior of regular expressions and the collation functions.

40500 *LC_CTYPE* Affects the behavior of regular expressions, character classification, character
40501 conversion functions, and wide-character functions.

40502 CX *LC_MESSAGES* Affects what strings are expected by commands and utilities as affirmative or
40503 negative responses.

40504 XSI It also affects what strings are given by commands and utilities as affirmative
40505 or negative responses, and the content of messages.

40506 *LC_MONETARY* Affects the behavior of functions that handle monetary values.

40507 *LC_NUMERIC* Affects the behavior of functions that handle numeric values.

40508 *LC_TIME* Affects the behavior of the time conversion functions.

40509 The *locale* argument is a pointer to a character string containing the required setting of *category*.
40510 The contents of this string are implementation-defined. In addition, the following preset values
40511 of *locale* are defined for all settings of *category*:

40512 CX "POSIX" Specifies the minimal environment for C-language translation called POSIX
40513 locale. If *setlocale*( ) is not invoked, the POSIX locale is the default at entry to
40514 *main*( ).

40515 "C" Equivalent to `"POSIX"`. |

40516 CX "" Specifies an implementation-defined native environment. This corresponds to |
40517 the value of the associated environment variables, *LC_** and *LANG*; see the |
40518 Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale and the
40519 Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment
40520 Variables. |

40521 A null pointer Used to direct *setlocale*( ) to query the current internationalized environment
40522 and return the name of the *locale*( ).

40523 THR The locale state is common to all threads within a process.

40524 **RETURN VALUE**

40525 Upon successful completion, *setlocale*( ) shall return the string associated with the specified
40526 category for the new locale. Otherwise, *setlocale*( ) shall return a null pointer and the program's
40527 locale is not changed.

A null pointer for *locale* causes *setlocale*( ) to return a pointer to the string associated with the *category* for the program's current locale. The program's locale shall not be changed.

The string returned by *setlocale*( ) is such that a subsequent call with that string and its associated *category* shall restore that part of the program's locale. The application shall not modify the string returned which may be overwritten by a subsequent call to *setlocale*( ).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The following code illustrates how a program can initialize the international environment for one language, while selectively modifying the program's locale such that regular expressions and string operations can be applied to text recorded in a different language:

```
setlocale(LC_ALL, "De");
setlocale(LC_COLLATE, "Fr@dict");
```

Internationalized programs must call *setlocale*( ) to initiate a specific language operation. This can be done by calling *setlocale*( ) as follows:

```
setlocale(LC_ALL, "");
```

Changing the setting of *LC_MESSAGES* has no effect on catalogs that have already been opened by calls to *catopen*( ).

RATIONALE

The ISO C standard defines a collection of functions to support internationalization. One of the most significant aspects of these functions is a facility to set and query the *international environment*. The international environment is a repository of information that affects the behavior of certain functionality, namely:

1. Character handling
2. Collating
3. Date / time formatting
4. Numeric editing
5. Monetary formatting
6. Messaging

The *setlocale*( ) function provides the application developer with the ability to set all or portions, called *categories*, of the international environment. These categories correspond to the areas of functionality, mentioned above. The syntax for *setlocale*( ) is as follows:

```
char *setlocale(int category, const char *locale);
```

where *category* is the name of one of following categories, namely:

LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME

In addition, a special value called *LC_ALL* directs *setlocale*( ) to set all categories.

There are two primary uses of *setlocale*( ):

1. Querying the international environment to find out what it is set to
2. Setting the international environment, or *locale*, to a specific value

The behavior of *setlocale*( ) in these two areas is described below. Since it is difficult to describe the behavior in words, examples are used to illustrate the behavior of specific uses.

To query the international environment, *setlocale*( ) is invoked with a specific category and the NULL pointer as the locale. The NULL pointer is a special directive to *setlocale*( ) that tells it to query rather than set the international environment. The following syntax is used to query the name of the international environment:

```
setlocale({LC_ALL, LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, \
    LC_NUMERIC, LC_TIME},(char *) NULL);
```

The *setlocale*( ) function shall return the string corresponding to the current international environment. This value may be used by a subsequent call to *setlocale*( ) to reset the international environment to this value. However, it should be noted that the return value from *setlocale*( ) may be a pointer to a static area within the function and is not guaranteed to remain unchanged (that is, it may be modified by a subsequent call to *setlocale*( )). Therefore, if the purpose of calling *setlocale*( ) is to save the value of the current international environment so it can be changed and reset later, the return value should be copied to an array of **char** in the calling program.

There are three ways to set the international environment with *setlocale*( ):

setlocale(*category*, *string*)
: This usage sets a specific *category* in the international environment to a specific value corresponding to the value of the *string*. A specific example is provided below:

```
setlocale(LC_ALL, "fr_FR.ISO-8859-1");
```

: In this example, all categories of the international environment are set to the locale corresponding to the string `"fr_FR.ISO-8859-1"`, or to the French language as spoken in France using the ISO / IEC 8859-1: 1998 standard codeset.

: If the string does not correspond to a valid locale, *setlocale*( ) shall return a NULL pointer and the international environment is not changed. Otherwise, *setlocale*( ) shall return the name of the locale just set.

setlocale(*category*, "C")
: The ISO C standard states that one locale must exist on all conforming implementations. The name of the locale is C and corresponds to a minimal international environment needed to support the C programming language.

setlocale(*category*, "")
: This sets a specific category to an implementation-defined default. This corresponds to the value of the environment variables.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *isalnum*( ), *isalpha*( ), *isblank*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *iswalnum*( ), *iswalpha*( ), *iswblank*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *isxdigit*( ),

localeconv( ), *mblen*( ), *mbstowcs*( ), *mbtowc*( ), *nl_langinfo*( ), *printf*( ), *scanf*( ), *setlocale*( ), *strcoll*( ), *strerror*( ), *strfmon*( ), *strtod*( ), *strxfrm*( ), *tolower*( ), *toupper*( ), *towlower*( ), *towupper*( ), *wcscoll*( ), *wcstod*( ), *wcstombs*( ), *wcsxfrm*( ), *wctomb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<langinfo.h>**, **<locale.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

Extensions beyond the ISO C standard are now marked.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

40625 **NAME**

40626 setlogmask — set log priority mask

40627 **SYNOPSIS**

40628 XSI
```
#include <syslog.h>

int setlogmask(int maskpri);
```
40629

40630

40631 **DESCRIPTION**

40632 Refer to *closelog*( ).

40633 **NAME**

40634 setnetent — network database function

40635 **SYNOPSIS**

40636 `#include <netdb.h>`

40637 `void setnetent(int stayopen);`

40638 **DESCRIPTION**

40639 Refer to *endnetent*( ).

40640 **NAME**

40641 setpgid — set process group ID for job control

40642 **SYNOPSIS**

40643 `#include <unistd.h>`

40644 `int setpgid(pid_t pid, pid_t pgid);`

40645 **DESCRIPTION**

40646 The *setpgid*( ) function shall either join an existing process group or create a new process group |
40647 within the session of the calling process. The process group ID of a session leader shall not
40648 change. Upon successful completion, the process group ID of the process with a process ID that
40649 matches *pid* shall be set to *pgid*. As a special case, if *pid* is 0, the process ID of the calling process
40650 shall be used. Also, if *pgid* is 0, the process group ID of the indicated process shall be used.

40651 **RETURN VALUE**

40652 Upon successful completion, *setpgid*( ) shall return 0; otherwise, –1 shall be returned and *errno*
40653 shall be set to indicate the error.

40654 **ERRORS**

40655 The *setpgid*( ) function shall fail if:

40656 [EACCES] The value of the *pid* argument matches the process ID of a child process of the
40657 calling process and the child process has successfully executed one of the *exec*
40658 functions.

40659 [EINVAL] The value of the *pgid* argument is less than 0, or is not a value supported by
40660 the implementation.

40661 [EPERM] The process indicated by the *pid* argument is a session leader.

40662 [EPERM] The value of the *pid* argument matches the process ID of a child process of the
40663 calling process and the child process is not in the same session as the calling
40664 process.

40665 [EPERM] The value of the *pgid* argument is valid but does not match the process ID of
40666 the process indicated by the *pid* argument and there is no process with a
40667 process group ID that matches the value of the *pgid* argument in the same
40668 session as the calling process.

40669 [ESRCH] The value of the *pid* argument does not match the process ID of the calling
40670 process or of a child process of the calling process.

40671 **EXAMPLES**

40672 None.

40673 **APPLICATION USAGE**

40674 None.

40675 **RATIONALE**

40676 The *setpgid*( ) function shall group processes together for the purpose of signaling, placement in |
40677 foreground or background, and other job control actions.

40678 The *setpgid*( ) function is similar to the *setpgrp*( ) function of 4.2 BSD, except that 4.2 BSD allowed
40679 the specified new process group to assume any value. This presents certain security problems
40680 and is more flexible than necessary to support job control.

40681 To provide tighter security, *setpgid*( ) only allows the calling process to join a process group
40682 already in use inside its session or create a new process group whose process group ID was
40683 equal to its process ID.

When a job control shell spawns a new job, the processes in the job must be placed into a new process group via *setpgid*( ). There are two timing constraints involved in this action:

1. The new process must be placed in the new process group before the appropriate program is launched via one of the *exec* functions.
2. The new process must be placed in the new process group before the shell can correctly send signals to the new process group.

To address these constraints, the following actions are performed. The new processes call *setpgid*( ) to alter their own process groups after *fork*( ) but before *exec*. This satisfies the first constraint. Under 4.3 BSD, the second constraint is satisfied by the synchronization property of *vfork*( ); that is, the shell is suspended until the child has completed the *exec*, thus ensuring that the child has completed the *setpgid*( ). A new version of *fork*( ) with this same synchronization property was considered, but it was decided instead to merely allow the parent shell process to adjust the process group of its child processes via *setpgid*( ). Both timing constraints are now satisfied by having both the parent shell and the child attempt to adjust the process group of the child process; it does not matter which succeeds first.

Since it would be confusing to an application to have its process group change after it began executing (that is, after *exec*), and because the child process would already have adjusted its process group before this, the [EACCES] error was added to disallow this.

One non-obvious use of *setpgid*( ) is to allow a job control shell to return itself to its original process group (the one in effect when the job control shell was executed). A job control shell does this before returning control back to its parent when it is terminating or suspending itself as a way of restoring its job control ‘‘state’’ back to what its parent would expect. (Note that the original process group of the job control shell typically matches the process group of its parent, but this is not necessarily always the case.)

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *getpgrp*( ), *setsid*( ), *tcsetpgrp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The *setpgid*( ) function is mandatory since _POSIX_JOB_CONTROL is required to be defined in this issue. This is a FIPS requirement.

40725 **NAME**

40726 setpgrp — set process group ID

40727 **SYNOPSIS**

40728 XSI
```
#include <unistd.h>
```

40729
```
pid_t setpgrp(void);
```

40730

40731 **DESCRIPTION**

40732 If the calling process is not already a session leader, *setpgrp*( ) sets the process group ID of the
40733 calling process to the process ID of the calling process. If *setpgrp*( ) creates a new session, then
40734 the new session has no controlling terminal.

40735 The *setpgrp*( ) function has no effect when the calling process is a session leader.

40736 **RETURN VALUE**

40737 Upon completion, *setpgrp*( ) shall return the process group ID.

40738 **ERRORS**

40739 No errors are defined.

40740 **EXAMPLES**

40741 None.

40742 **APPLICATION USAGE**

40743 None.

40744 **RATIONALE**

40745 None.

40746 **FUTURE DIRECTIONS**

40747 None.

40748 **SEE ALSO**

40749 *exec*, *fork*( ), *getpid*( ), *getsid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of
40750 IEEE Std 1003.1-200x, **<unistd.h>**

40751 **CHANGE HISTORY**

40752 First released in Issue 4, Version 2.

40753 **Issue 5**

40754 Moved from X/OPEN UNIX extension to BASE.

40755 **NAME**

40756 setpriority — set the nice value

40757 **SYNOPSIS**

40758 XSI
```
#include <sys/resource.h>
```

40759
```
int setpriority(int which, id_t who, int nice);
```

40760

40761 **DESCRIPTION**

40762 Refer to *getpriority*( ).

40763 **NAME**

40764 setprotoent — network protocol database functions

40765 **SYNOPSIS**

40766 `#include <netdb.h>`

40767 `void setprotoent(int stayopen);`

40768 **DESCRIPTION**

40769 Refer to *endprotoent*( ).

40770 **NAME**

40771 setpwent — user database function

40772 **SYNOPSIS**

40773 XSI
```
#include <pwd.h>
```
40774
```
void setpwent(void);
```
40775

40776 **DESCRIPTION**

40777 Refer to *endpwent*( ).

40778 **NAME**

40779 setregid — set real and effective group IDs

40780 **SYNOPSIS**

40781 XSI
```
#include <unistd.h>

int setregid(gid_t rgid, gid_t egid);
```
40782

40783

40784 **DESCRIPTION**

40785 The *setregid*( ) function shall set the real and effective group IDs of the calling process.

40786 If *rgid* is –1, the real group ID shall not be changed; if *egid* is –1, the effective group ID shall not
40787 be changed.

40788 The real and effective group IDs may be set to different values in the same call.

40789 Only a process with appropriate privileges can set the real group ID and the effective group ID
40790 to any valid value.

40791 A non-privileged process can set either the real group ID to the saved set-group-ID from one of
40792 the *exec* family of functions, or the effective group ID to the saved set-group-ID or the real group
40793 ID.

40794 Any supplementary group IDs of the calling process remain unchanged.

40795 **RETURN VALUE**

40796 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
40797 indicate the error, and neither of the group IDs are changed.

40798 **ERRORS**

40799 The *setregid*( ) function shall fail if:

40800 [EINVAL] The value of the *rgid* or *egid* argument is invalid or out-of-range.

40801 [EPERM] The process does not have appropriate privileges and a change other than
40802 changing the real group ID to the saved set-group-ID, or changing the
40803 effective group ID to the real group ID or the saved set-group-ID, was
40804 requested.

40805 **EXAMPLES**

40806 None.

40807 **APPLICATION USAGE**

40808 If a set-group-ID process sets its effective group ID to its real group ID, it can still set its effective
40809 group ID back to the saved set-group-ID.

40810 **RATIONALE**

40811 None.

40812 **FUTURE DIRECTIONS**

40813 None.

40814 **SEE ALSO**

40815 *exec*, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setreuid*( ), *setuid*( ), the
40816 Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

40817 **CHANGE HISTORY**

40818 First released in Issue 4, Version 2.

40819 **Issue 5**

40820 Moved from X/OPEN UNIX extension to BASE.

40821 The DESCRIPTION is updated to indicate that the saved set-group-ID can be set by any of the
40822 *exec* family of functions, not just *execev*( ).

40823 **NAME**

40824 setreuid — set real and effective user IDs

40825 **SYNOPSIS**

40826 XSI
```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);
```
40827

40828

40829 **DESCRIPTION**

40830 The *setreuid*( ) function shall set the real and effective user IDs of the current process to the | 40831 values specified by the *ruid* and *euid* arguments. If *ruid* or *euid* is –1, the corresponding effective | 40832 or real user ID of the current process shall be left unchanged. |

40833 A process with appropriate privileges can set either ID to any value. An unprivileged process 40834 can only set the effective user ID if the *euid* argument is equal to either the real, effective, or 40835 saved user ID of the process.

40836 It is unspecified whether a process without appropriate privileges is permitted to change the real 40837 user ID to match the current real, effective, or saved set-user-ID of the process.

40838 **RETURN VALUE**

40839 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to 40840 indicate the error.

40841 **ERRORS**

40842 The *setreuid*( ) function shall fail if:

40843 [EINVAL] The value of the *ruid* or *euid* argument is invalid or out-of-range.

40844 [EPERM] The current process does not have appropriate privileges, and either an 40845 attempt was made to change the effective user ID to a value other than the 40846 real user ID or the saved set-user-ID or an attempt was made to change the 40847 real user ID to a value not permitted by the implementation.

40848 **EXAMPLES**

40849 **Setting the Effective User ID to the Real User ID**

40850 The following example sets the effective user ID of the calling process to the real user ID, so that 40851 files created later will be owned by the current user.

```
40852 #include <unistd.h>
40853 #include <sys/types.h>
40854 ...
40855 setreuid(getuid(), getuid());
40856 ...
```

40857 **APPLICATION USAGE**

40858 None.

40859 **RATIONALE**

40860 None.

40861 **FUTURE DIRECTIONS**

40862 None.

40863 **SEE ALSO**

40864 *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setuid*( ), the Base
40865 Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

40866 **CHANGE HISTORY**

40867 First released in Issue 4, Version 2.

40868 **Issue 5**

40869 Moved from X/OPEN UNIX extension to BASE.

40870 **NAME**

40871 setrlimit — control maximum resource consumption

40872 **SYNOPSIS**

40873 XSI
```
#include <sys/resource.h>
```

40874
```
int setrlimit(int resource, const struct rlimit *rlp);
```

40875

40876 **DESCRIPTION**

40877 Refer to *getrlimit*( ).

40878 **NAME**

40879 setservent — network services database functions

40880 **SYNOPSIS**

40881 `#include <netdb.h>`

40882 `void setservent(int stayopen);`

40883 **DESCRIPTION**

40884 Refer to *endservent*( ).

40885 **NAME**

40886 setsid — create session and set process group ID

40887 **SYNOPSIS**

40888
```
#include <unistd.h>
```

40889
```
pid_t setsid(void);
```

40890 **DESCRIPTION**

40891 The *setsid*( ) function shall create a new session, if the calling process is not a process group
40892 leader. Upon return the calling process shall be the session leader of this new session, shall be
40893 the process group leader of a new process group, and shall have no controlling terminal. The
40894 process group ID of the calling process shall be set equal to the process ID of the calling process.
40895 The calling process shall be the only process in the new process group and the only process in
40896 the new session.

40897 **RETURN VALUE**

40898 Upon successful completion, *setsid*( ) shall return the value of the new process group ID of the
40899 calling process. Otherwise, it shall return (**pid_t**)–1 and set *errno* to indicate the error.

40900 **ERRORS**

40901 The *setsid*( ) function shall fail if:

40902 [EPERM] The calling process is already a process group leader, or the process group ID
40903 of a process other than the calling process matches the process ID of the
40904 calling process.

40905 **EXAMPLES**

40906 None.

40907 **APPLICATION USAGE**

40908 None.

40909 **RATIONALE**

40910 The *setsid*( ) function is similar to the *setpgrp*( ) function of System V. System V, without job
40911 control, groups processes into process groups and creates new process groups via *setpgrp*( ); only
40912 one process group may be part of a login session.

40913 Job control allows multiple process groups within a login session. In order to limit job control
40914 actions so that they can only affect processes in the same login session, this volume of
40915 IEEE Std 1003.1-200x adds the concept of a session that is created via *setsid*( ). The *setsid*( )
40916 function also creates the initial process group contained in the session. Additional process
40917 groups can be created via the *setpgid*( ) function. A System V process group would correspond to
40918 a POSIX System Interfaces session containing a single POSIX process group. Note that this
40919 function requires that the calling process not be a process group leader. The usual way to ensure
40920 this is true is to create a new process with *fork*( ) and have it call *setsid*( ). The *fork*( ) function
40921 guarantees that the process ID of the new process does not match any existing process group ID.

40922 **FUTURE DIRECTIONS**

40923 None.

40924 **SEE ALSO**

40925 *getsid*( ), *setpgid*( ), *setpgrp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**,
40926 **<unistd.h>**

40927 **CHANGE HISTORY**

40928 First released in Issue 3.

40929 Entry included for alignment with the POSIX.1-1988 standard.

40930 **Issue 6**

40931 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

40932 The following new requirements on POSIX implementations derive from alignment with the
40933 Single UNIX Specification:

40934 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
40935 required for conforming implementations of previous POSIX specifications, it was not
40936 required for UNIX applications.

40937 **NAME**

40938 setsockopt — set the socket options

40939 **SYNOPSIS**

40940 `#include <sys/socket.h>`

40941 `int setsockopt(int socket, int level, int option_name,`
40942 `    const void *option_value, socklen_t option_len);`

40943 **DESCRIPTION**

40944 The *setsockopt*( ) function shall set the option specified by the *option_name* argument, at the |
40945 protocol level specified by the *level* argument, to the value pointed to by the *option_value*
40946 argument for the socket associated with the file descriptor specified by the *socket* argument.

40947 The *level* argument specifies the protocol level at which the option resides. To set options at the
40948 socket level, specify the *level* argument as SOL_SOCKET. To set options at other levels, supply
40949 the appropriate *level* identifier for the protocol controlling the option. For example, to indicate
40950 that an option is interpreted by the TCP (Transport Control Protocol), set *level* to IPPROTO_TCP
40951 as defined in the **<netinet/in.h>** header.

40952 The *option_name* argument specifies a single option to set. The *option_name* argument and any
40953 specified options are passed uninterpreted to the appropriate protocol module for
40954 interpretations. The **<sys/socket.h>** header defines the socket-level options. The options are as
40955 follows:

40956 SO_DEBUG — Turns on recording of debugging information. This option enables or
40957 disables debugging in the underlying protocol modules. This option takes
40958 an **int** value. This is a Boolean option.

40959 SO_BROADCAST — Permits sending of broadcast messages, if this is supported by the
40960 protocol. This option takes an **int** value. This is a Boolean option.

40961 SO_REUSEADDR — Specifies that the rules used in validating addresses supplied to *bind*( )
40962 should allow reuse of local addresses, if this is supported by the protocol.
40963 This option takes an **int** value. This is a Boolean option.

40964 SO_KEEPALIVE — Keeps connections active by enabling the periodic transmission of
40965 messages, if this is supported by the protocol. This option takes an **int**
40966 value.

40967 If the connected socket fails to respond to these messages, the connection
40968 is broken and threads writing to that socket are notified with a SIGPIPE
40969 signal.

40970 This is a Boolean option.

40971 SO_LINGER — Lingers on a *close*( ) if data is present. This option controls the action
40972 taken when unsent messages queue on a socket and *close*( ) is performed. |
40973 If SO_LINGER is set, the system shall block the process during *close*( ) |
40974 until it can transmit the data or until the time expires. If SO_LINGER is
40975 not specified, and *close*( ) is issued, the system handles the call in a way
40976 that allows the process to continue as quickly as possible. This option
40977 takes a **linger** structure, as defined in the **<sys/socket.h>** header, to
40978 specify the state of the option and linger interval.

40979 SO_OOBINLINE — Leaves received out-of-band data (data marked urgent) inline. This
40980 option takes an **int** value. This is a Boolean option.

SO_SNDBUF Sets send buffer size. This option takes an **int** value.

SO_RCVBUF Sets receive buffer size. This option takes an **int** value.

SO_DONTROUTE Requests that outgoing messages bypass the standard routing facilities. The destination shall be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option takes an **int** value. This is a Boolean option.

SO_RCVLOWAT Sets the minimum number of bytes to process for socket input operations. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. (They may return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that returned; for example, out-of-band data.) This option takes an **int** value. Note that not all implementations allow this option to be set.

SO_RCVTIMEO Sets the timeout value that specifies the maximum amount of time an input function waits until it completes. It accepts a **timeval** structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it shall return with a partial count or *errno* set to [EAGAIN] or [EWOULDBLOCK] if no data is received. The default for this option is zero, which indicates that a receive operation shall not time out. This option takes a **timeval** structure. Note that not all implementations allow this option to be set.

SO_SNDLOWAT Sets the minimum number of bytes to process for socket output operations. Non-blocking output operations shall process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. This option takes an **int** value. Note that not all implementations allow this option to be set.

SO_SNDTIMEO Sets the timeout value specifying the amount of time that an output function blocks because flow control prevents data from being sent. If a send operation has blocked for this time, it shall return with a partial count or with *errno* set to [EAGAIN] or [EWOULDBLOCK] if no data is sent. The default for this option is zero, which indicates that a send operation shall not time out. This option stores a **timeval** structure. Note that not all implementations allow this option to be set.

For Boolean options, 0 indicates that the option is disabled and 1 indicates that the option is enabled.

Options at other protocol levels vary in format and name.

RETURN VALUE

Upon successful completion, *setsockopt*( ) shall return 0. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *setsockopt*( ) function shall fail if:

[EBADF] The *socket* argument is not a valid file descriptor.

41027 [EDOM] The send and receive timeout values are too big to fit into the timeout fields in
41028 the socket structure.

41029 [EINVAL] The specified option is invalid at the specified socket level or the socket has
41030 been shut down.

41031 [EISCONN] The socket is already connected, and a specified option cannot be set while the
41032 socket is connected.

41033 [ENOPROTOOPT]
41034 The option is not supported by the protocol.

41035 [ENOTSOCK] The *socket* argument does not refer to a socket.

41036 The *setsockopt*( ) function may fail if:

41037 [ENOMEM] There was insufficient memory available for the operation to complete.

41038 [ENOBUFS] Insufficient resources are available in the system to complete the call.

41039 **EXAMPLES**

41040 None.

41041 **APPLICATION USAGE**

41042 The *setsockopt*( ) function provides an application program with the means to control socket
41043 behavior. An application program can use *setsockopt*( ) to allocate buffer space, control timeouts,
41044 or permit socket data broadcasts. The **<sys/socket.h>** header defines the socket-level options
41045 available to *setsockopt*( ).

41046 Options may exist at multiple protocol levels. The SO_ options are always present at the
41047 uppermost socket level.

41048 **RATIONALE**

41049 None.

41050 **FUTURE DIRECTIONS**

41051 None.

41052 **SEE ALSO**

41053 Section 2.10 (on page 508), *bind*( ), *endprotoent*( ), *getsockopt*( ), *socket*( ), the Base Definitions
41054 volume of IEEE Std 1003.1-200x, **<netinet/in.h>**, **<sys/socket.h>**

41055 **CHANGE HISTORY**

41056 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

41057 **NAME**

41058 setstate — switch pseudo-random number generator state arrays

41059 **SYNOPSIS**

41060 XSI
```
#include <stdlib.h>
```

41061
```
char *setstate(const char *state);
```

41062

41063 **DESCRIPTION**

41064 Refer to *initstate*( ).

41065 **NAME**

41066 setuid — set user ID

41067 **SYNOPSIS**

41068 `#include <unistd.h>`

41069 `int setuid(uid_t uid);`

41070 **DESCRIPTION**

41071 If the process has appropriate privileges, *setuid*( ) shall set the real user ID, effective user ID, and
41072 the saved set-user-ID of the calling process to *uid*.

41073 If the process does not have appropriate privileges, but *uid* is equal to the real user ID or the
41074 saved set-user-ID, *setuid*( ) shall set the effective user ID to *uid*; the real user ID and saved set-
41075 user-ID shall remain unchanged.

41076 The *setuid*( ) function shall not affect the supplementary group list in any way.

41077 **RETURN VALUE**

41078 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
41079 indicate the error.

41080 **ERRORS**

41081 The *setuid*( ) function shall fail, return –1, and set *errno* to the corresponding value if one or more
41082 of the following are true:

41083 [EINVAL] The value of the *uid* argument is invalid and not supported by the
41084 implementation.

41085 [EPERM] The process does not have appropriate privileges and *uid* does not match the
41086 real user ID or the saved set-user-ID.

41087 **EXAMPLES**

41088 None.

41089 **APPLICATION USAGE**

41090 None.

41091 **RATIONALE**

41092 The various behaviors of the *setuid*( ) and *setgid*( ) functions when called by non-privileged
41093 processes reflect the behavior of different historical implementations. For portability, it is
41094 recommended that new non-privileged applications use the *seteuid*( ) and *setegid*( ) functions
41095 instead.

41096 The saved set-user-ID capability allows a program to regain the effective user ID established at
41097 the last *exec* call. Similarly, the saved set-group-ID capability allows a program to regain the
41098 effective group ID established at the last *exec* call. These capabilities are derived from System V.
41099 Without them, a program might have to run as superuser in order to perform the same |
41100 functions, because superuser can write on the user's files. This is a problem because such a |
41101 program can write on any user's files, and so must be carefully written to emulate the |
41102 permissions of the calling process properly. In System V, these capabilities have traditionally
41103 been implemented only via the *setuid*( ) and *setgid*( ) functions for non-privileged processes. The
41104 fact that the behavior of those functions was different for privileged processes made them
41105 difficult to use. The POSIX.1-1990 standard defined the *setuid*( ) function to behave differently
41106 for privileged and unprivileged users. When the caller had the appropriate privilege, the
41107 function set the calling process' real user ID, effective user ID, and saved set-user ID on |
41108 implementations that supported it. When the caller did not have the appropriate privilege, the
41109 function set only the effective user ID, subject to permission checks. The former use is generally
41110 needed for utilities like *login* and *su*, which are not conforming applications and thus outside the |

scope of IEEE Std 1003.1-200x. These utilities wish to change the user ID irrevocably to a new value, generally that of an unprivileged user. The latter use is needed for conforming applications that are installed with the set-user-ID bit and need to perform operations using the real user ID.

IEEE Std 1003.1-200x augments the latter functionality with a mandatory feature named _POSIX_SAVED_IDS. This feature permits a set-user-ID application to switch its effective user ID back and forth between the values of its *exec*-time real user ID and effective user ID. Unfortunately, the POSIX.1-1990 standard did not permit a conforming application using this feature to work properly when it happened to be executed with the (implementation-defined) appropriate privilege. Furthermore, the application did not even have a means to tell whether it had this privilege. Since the saved set-user-ID feature is quite desirable for applications, as evidenced by the fact that NIST required it in FIPS 151-2, it has been mandated by IEEE Std 1003.1-200x. However, there are implementors who have been reluctant to support it given the limitation described above.

The 4.3BSD system handles the problem by supporting separate functions: *setuid*() (which always sets both the real and effective user IDs, like *setuid*() in IEEE Std 1003.1-200x for privileged users), and *seteuid*() (which always sets just the effective user ID, like *setuid*() in IEEE Std 1003.1-200x for non-privileged users). This separation of functionality into distinct functions seems desirable. 4.3BSD does not support the saved set-user-ID feature. It supports similar functionality of switching the effective user ID back and forth via *setreuid*(), which permits reversing the real and effective user IDs. This model seems less desirable than the saved set-user-ID because the real user ID changes as a side effect. The current 4.4BSD includes saved effective IDs and uses them for *seteuid*() and *setegid*() as described above. The *setreuid*() and *setregid*() functions will be deprecated or removed.

The solution here is:

- Require that all implementations support the functionality of the saved set-user-ID, which is set by the *exec* functions and by privileged calls to *setuid*().
- Add the *seteuid*() and *setegid*() functions as portable alternatives to *setuid*() and *setgid*() for non-privileged and privileged processes.

Historical systems have provided two mechanisms for a set-user-ID process to change its effective user ID to be the same as its real user ID in such a way that it could return to the original effective user ID: the use of the *setuid*() function in the presence of a saved set-user-ID, or the use of the BSD *setreuid*() function, which was able to swap the real and effective user IDs. The changes included in IEEE Std 1003.1-200x provide a new mechanism using *seteuid*() in conjunction with a saved set-user-ID. Thus, all implementations with the new *seteuid*() mechanism will have a saved set-user-ID for each process, and most of the behavior controlled by _POSIX_SAVED_IDS has been changed to agree with the case where the option was defined. The *kill*() function is an exception. Implementors of the new *seteuid*() mechanism will generally be required to maintain compatibility with the older mechanisms previously supported by their systems. However, compatibility with this use of *setreuid*() and with the _POSIX_SAVED_IDS behavior of *kill*() is unfortunately complicated. If an implementation with a saved set-user-ID allows a process to use *setreuid*() to swap its real and effective user IDs, but were to leave the saved set-user-ID unmodified, the process would then have an effective user ID equal to the original real user ID, and both real and saved set-user-ID would be equal to the original effective user ID. In that state, the real user would be unable to kill the process, even though the effective user ID of the process matches that of the real user, if the *kill*() behavior of _POSIX_SAVED_IDS was used. This is obviously not acceptable. The alternative choice, which is used in at least one implementation, is to change the saved set-user-ID to the effective user ID during most calls to *setreuid*(). The standard developers considered that alternative to be less correct than the

retention of the old behavior of *kill*() in such systems. Current conforming applications shall accommodate either behavior from *kill*(), and there appears to be no strong reason for *kill*() to check the saved set-user-ID rather than the effective user ID.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *getegid*(), *geteuid*(), *getgid*(), *getuid*(), *setegid*(), *seteuid*(), *setgid*(), *setregid*(), *setreuid*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The functionality associated with _POSIX_SAVED_IDS is now mandatory. This is a FIPS requirement.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effects of *setuid*() in processes without appropriate privileges are changed.
- A requirement that the supplementary group list is not affected is added.

41182 **NAME**

41183 setutxent — reset user accounting database to first entry

41184 **SYNOPSIS**

41185 XSI
```
#include <utmpx.h>

void setutxent(void);
```
41186

41187

41188 **DESCRIPTION**

41189 Refer to *endutxent*( ).

41190 **NAME**

41191 setvbuf — assign buffering to a stream

41192 **SYNOPSIS**

41193
```
#include <stdio.h>
```

41194
```
int setvbuf(FILE *restrict stream, char *restrict buf, int type,
```
41195
```
    size_t size);
```

41196 **DESCRIPTION**

41197 CX The functionality described on this reference page is aligned with the ISO C standard. Any
41198 conflict between the requirements described here and the ISO C standard is unintentional. This
41199 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

41200 The *setvbuf*( ) function may be used after the stream pointed to by *stream* is associated with an
41201 open file but before any other operation (other than an unsuccessful call to *setvbuf*( )) is
41202 performed on the stream. The argument *type* determines how *stream* shall be buffered, as
41203 follows:

41204 • {_IOFBF} shall cause input/output to be fully buffered.

41205 • {_IOLBF} shall cause input/output to be line buffered.

41206 • {_IONBF} shall cause input/output to be unbuffered.

41207 If *buf* is not a null pointer, the array it points to may be used instead of a buffer allocated by
41208 *setvbuf*( ) and the argument *size* specifies the size of the array; otherwise, *size* may determine the
41209 size of a buffer allocated by the *setvbuf*( ) function. The contents of the array at any time are |
41210 unspecified. |

41211 For information about streams, see Section 2.5 (on page 484).

41212 **RETURN VALUE**

41213 Upon successful completion, *setvbuf*( ) shall return 0. Otherwise, it shall return a non-zero value
41214 CX if an invalid value is given for *type* or if the request cannot be honored, and may set *errno* to
41215 indicate the error.

41216 **ERRORS**

41217 The *setvbuf*( ) function may fail if:

41218 CX [EBADF] The file descriptor underlying *stream* is not valid.

41219 **EXAMPLES**

41220 None.

41221 **APPLICATION USAGE**

41222 A common source of error is allocating buffer space as an ‘‘automatic’’ variable in a code block,
41223 and then failing to close the stream in the same block.

41224 With *setvbuf*( ), allocating a buffer of *size* bytes does not necessarily imply that all of *size* bytes are
41225 used for the buffer area.

41226 Applications should note that many implementations only provide line buffering on input from
41227 terminal devices.

41228 **RATIONALE**

41229 None.

41230 **FUTURE DIRECTIONS**

41231 None.

41232 **SEE ALSO**

41233 *fopen*( ), *setbuf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

41234 **CHANGE HISTORY**

41235 First released in Issue 1. Derived from Issue 1 of the SVID.

41236 **Issue 6**

41237 Extensions beyond the ISO C standard are now marked.

41238 The *setvbuf*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

41239 **NAME**

41240 shm_open — open a shared memory object (**REALTIME**)

41241 **SYNOPSIS**

41242 SHM
```
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);
```
41243

41244

41245 **DESCRIPTION**

41246 The *shm_open*( ) function shall establish a connection between a shared memory object and a file
41247 descriptor. It shall create an open file description that refers to the shared memory object and a
41248 file descriptor that refers to that open file description. The file descriptor is used by other
41249 functions to refer to that shared memory object. The *name* argument points to a string naming a
41250 shared memory object. It is unspecified whether the name appears in the file system and is
41251 visible to other functions that take pathnames as arguments. The *name* argument conforms to the |
41252 construction rules for a pathname. If *name* begins with the slash character, then processes calling |
41253 *shm_open*( ) with the same value of *name* refer to the same shared memory object, as long as that
41254 name has not been removed. If *name* does not begin with the slash character, the effect is
41255 implementation-defined. The interpretation of slash characters other than the leading slash
41256 character in *name* is implementation-defined.

41257 If successful, *shm_open*( ) shall return a file descriptor for the shared memory object that is the
41258 lowest numbered file descriptor not currently open for that process. The open file description is
41259 new, and therefore the file descriptor does not share it with any other processes. It is unspecified
41260 whether the file offset is set. The FD_CLOEXEC file descriptor flag associated with the new file
41261 descriptor is set.

41262 The file status flags and file access modes of the open file description are according to the value
41263 of *oflag*. The *oflag* argument is the bitwise-inclusive OR of the following flags defined in the
41264 **<fcntl.h>** header. Applications specify exactly one of the first two values (access modes) below
41265 in the value of *oflag*:

41266 O_RDONLY Open for read access only.

41267 O_RDWR Open for read or write access.

41268 Any combination of the remaining flags may be specified in the value of *oflag*:

41269 O_CREAT If the shared memory object exists, this flag has no effect, except as noted
41270 under O_EXCL below. Otherwise, the shared memory object is created; the
41271 user ID of the shared memory object shall be set to the effective user ID of the
41272 process; the group ID of the shared memory object is set to a system default
41273 group ID or to the effective group ID of the process. The permission bits of the
41274 shared memory object shall be set to the value of the *mode* argument except
41275 those set in the file mode creation mask of the process. When bits in *mode*
41276 other than the file permission bits are set, the effect is unspecified. The *mode*
41277 argument does not affect whether the shared memory object is opened for
41278 reading, for writing, or for both. The shared memory object has a size of zero.

41279 O_EXCL If O_EXCL and O_CREAT are set, *shm_open*( ) fails if the shared memory
41280 object exists. The check for the existence of the shared memory object and the
41281 creation of the object if it does not exist is atomic with respect to other
41282 processes executing *shm_open*( ) naming the same shared memory object with
41283 O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the
41284 result is undefined.

41285 O_TRUNC If the shared memory object exists, and it is successfully opened O_RDWR,
41286 the object shall be truncated to zero length and the mode and owner shall be
41287 unchanged by this function call. The result of using O_TRUNC with
41288 O_RDONLY is undefined.

41289 When a shared memory object is created, the state of the shared memory object, including all
41290 data associated with the shared memory object, persists until the shared memory object is
41291 unlinked and all other references are gone. It is unspecified whether the name and shared
41292 memory object state remain valid after a system reboot.

41293 **RETURN VALUE**

41294 Upon successful completion, the *shm_open*( ) function shall return a non-negative integer
41295 representing the lowest numbered unused file descriptor. Otherwise, it shall return –1 and set
41296 *errno* to indicate the error.

41297 **ERRORS**

41298 The *shm_open*( ) function shall fail if:

41299 [EACCES] The shared memory object exists and the permissions specified by *oflag* are
41300 denied, or the shared memory object does not exist and permission to create
41301 the shared memory object is denied, or O_TRUNC is specified and write
41302 permission is denied.

41303 [EEXIST] O_CREAT and O_EXCL are set and the named shared memory object already
41304 exists.

41305 [EINTR] The *shm_open*( ) operation was interrupted by a signal.

41306 [EINVAL] The *shm_open*( ) operation is not supported for the given name.

41307 [EMFILE] Too many file descriptors are currently in use by this process.

41308 [ENAMETOOLONG]
41309 The length of the *name* argument exceeds {PATH_MAX} or a pathname |
41310 component is longer than {NAME_MAX}. |

41311 [ENFILE] Too many shared memory objects are currently open in the system.

41312 [ENOENT] O_CREAT is not set and the named shared memory object does not exist.

41313 [ENOSPC] There is insufficient space for the creation of the new shared memory object.

41314 **EXAMPLES**

41315 None.

41316 **APPLICATION USAGE**

41317 None.

41318 **RATIONALE**

41319 When the Memory Mapped Files option is supported, the normal *open*( ) call is used to obtain a
41320 descriptor to a file to be mapped according to existing practice with *mmap*( ). When the Shared
41321 Memory Objects option is supported, the *shm_open*( ) function shall obtain a descriptor to the |
41322 shared memory object to be mapped. |

41323 There is ample precedent for having a file descriptor represent several types of objects. In the
41324 POSIX.1-1990 standard, a file descriptor can represent a file, a pipe, a FIFO, a tty, or a directory.
41325 Many implementations simply have an operations vector, which is indexed by the file descriptor
41326 type and does very different operations. Note that in some cases the file descriptor passed to
41327 generic operations on file descriptors are returned by *open*( ) or *creat*( ) and in some cases
41328 returned by alternate functions, such as *pipe*( ). The latter technique is used by *shm_open*( ).

Note that such shared memory objects can actually be implemented as mapped files. In both cases, the size can be set after the open using *ftruncate*(). The *shm_open*() function itself does not create a shared object of a specified size because this would duplicate an extant function that set the size of an object referenced by a file descriptor.

On implementations where memory objects are implemented using the existing file system, the *shm_open*() function may be implemented using a macro that invokes *open*(), and the *shm_unlink*() function may be implemented using a macro that invokes *unlink*().

For implementations without a permanent file system, the definition of the name of the memory objects is allowed not to survive a system reboot. Note that this allows systems with a permanent file system to implement memory objects as data structures internal to the implementation as well.

On implementations that choose to implement memory objects using memory directly, a *shm_open*() followed by a *ftruncate*() and *close*() can be used to preallocate a shared memory area and to set the size of that preallocation. This may be necessary for systems without virtual memory hardware support in order to ensure that the memory is contiguous.

The set of valid open flags to *shm_open*() was restricted to O_RDONLY, O_RDWR, O_CREAT, and O_TRUNC because these could be easily implemented on most memory mapping systems. This volume of IEEE Std 1003.1-200x is silent on the results if the implementation cannot supply the requested file access because of implementation-defined reasons, including hardware ones.

The error conditions [EACCES] and [ENOTSUP] are provided to inform the application that the implementation cannot complete a request.

[EACCES] indicates for implementation-defined reasons, probably hardware-related, that the implementation cannot comply with a requested mode because it conflicts with another requested mode. An example might be that an application desires to open a memory object two times, mapping different areas with different access modes. If the implementation cannot map a single area into a process space in two places, which would be required if different access modes were required for the two areas, then the implementation may inform the application at the time of the second open.

[ENOTSUP] indicates for implementation-defined reasons, probably hardware-related, that the implementation cannot comply with a requested mode at all. An example would be that the hardware of the implementation cannot support write-only shared memory areas.

On all implementations, it may be desirable to restrict the location of the memory objects to specific file systems for performance (such as a RAM disk) or implementation-defined reasons (shared memory supported directly only on certain file systems). The *shm_open*() function may be used to enforce these restrictions. There are a number of methods available to the application to determine an appropriate name of the file or the location of an appropriate directory. One way is from the environment via *getenv*(). Another would be from a configuration file.

This volume of IEEE Std 1003.1-200x specifies that memory objects have initial contents of zero when created. This is consistent with current behavior for both files and newly allocated memory. For those implementations that use physical memory, it would be possible that such implementations could simply use available memory and give it to the process uninitialized. This, however, is not consistent with standard behavior for the uninitialized data area, the stack, and of course, files. Finally, it is highly desirable to set the allocated memory to zero for security reasons. Thus, initializing memory objects to zero is required.

41373 **FUTURE DIRECTIONS**

41374 None.

41375 **SEE ALSO**

41376 *close*( ), *dup*( ), *exec*, *fcntl*( ), *mmap*( ), *shmat*( ), *shmctl*( ), *shmdt*( ), *shm_unlink*( ), *umask*( ), the Base
41377 Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<sys/mman.h>**

41378 **CHANGE HISTORY**

41379 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

41380 **Issue 6**

41381 The *shm_open*( ) function is marked as part of the Shared Memory Objects option.

41382 The [ENOSYS] error condition has been removed as stubs need not be provided if an
41383 implementation does not support the Shared Memory Objects option.

41384 **NAME**

41385 shm_unlink — remove a shared memory object (**REALTIME**)

41386 **SYNOPSIS**

41387 SHM
```
#include <sys/mman.h>
```

41388
```
int shm_unlink(const char *name);
```

41389

41390 **DESCRIPTION**

41391 The *shm_unlink*( ) function shall remove the name of the shared memory object named by the
41392 string pointed to by *name*.

41393 If one or more references to the shared memory object exist when the object is unlinked, the
41394 name shall be removed before *shm_unlink*( ) returns, but the removal of the memory object
41395 contents shall be postponed until all open and map references to the shared memory object have
41396 been removed.

41397 Even if the object continues to exist after the last *shm_unlink*( ), reuse of the name shall
41398 subsequently cause *shm_open*( ) to behave as if no shared memory object of this name exists (that
41399 is, *shm_open*( ) will fail if O_CREAT is not set, or will create a new shared memory object if
41400 O_CREAT is set).

41401 **RETURN VALUE**

41402 Upon successful completion, a value of zero shall be returned. Otherwise, a value of –1 shall be
41403 returned and *errno* set to indicate the error. If –1 is returned, the named shared memory object
41404 shall not be changed by this function call.

41405 **ERRORS**

41406 The *shm_unlink*( ) function shall fail if:

41407 [EACCES] Permission is denied to unlink the named shared memory object.

41408 [ENAMETOOLONG]
41409 The length of the *name* argument exceeds {PATH_MAX} or a pathname |
41410 component is longer than {NAME_MAX}. |

41411 [ENOENT] The named shared memory object does not exist.

41412 **EXAMPLES**

41413 None.

41414 **APPLICATION USAGE**

41415 Names of memory objects that were allocated with *open*( ) are deleted with *unlink*( ) in the usual
41416 fashion. Names of memory objects that were allocated with *shm_open*( ) are deleted with
41417 *shm_unlink*( ). Note that the actual memory object is not destroyed until the last close and
41418 unmap on it have occurred if it was already in use.

41419 **RATIONALE**

41420 None.

41421 **FUTURE DIRECTIONS**

41422 None.

41423 **SEE ALSO**

41424 *close*( ), *mmap*( ), *munmap*( ), *shmat*( ), *shmctl*( ), *shmdt*( ), *shm_open*( ), the Base Definitions volume
41425 of IEEE Std 1003.1-200x, **<sys/mman.h>**

41426 **CHANGE HISTORY**

41427 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

41428 **Issue 6**

41429 The *shm_unlink*( ) function is marked as part of the Shared Memory Objects option.

41430 In the DESCRIPTION, text is added to clarify that reusing the same name after a *shm_unlink*( )
41431 will not attach to the old shared memory object.

41432 The [ENOSYS] error condition has been removed as stubs need not be provided if an
41433 implementation does not support the Shared Memory Objects option.

41434 **NAME**

41435 shmat — XSI shared memory attach operation

41436 **SYNOPSIS**

41437 XSI
```
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```
41438
41439

41440 **DESCRIPTION**

41441 The *shmat*( ) function operates on XSI shared memory (see the Base Definitions volume of
41442 IEEE Std 1003.1-200x, Section 3.340, Shared Memory Object). It is unspecified whether this
41443 function interoperates with the realtime interprocess communication facilities defined in Section
41444 2.8 (on page 491).

41445 The *shmat*( ) function attaches the shared memory segment associated with the shared memory
41446 identifier specified by *shmid* to the address space of the calling process. The segment is attached
41447 at the address specified by one of the following criteria:

41448 • If *shmaddr* is a null pointer, the segment is attached at the first available address as selected
41449 by the system.

41450 • If *shmaddr* is not a null pointer and (*shmflg* &SHM_RND) is non-zero, the segment is attached
41451 at the address given by (*shmaddr* –((*uintptr_t*)*shmaddr* %SHMLBA)). The character '%' is the
41452 C-language remainder operator.

41453 • If *shmaddr* is not a null pointer and (*shmflg* &SHM_RND) is 0, the segment is attached at the
41454 address given by *shmaddr*.

41455 • The segment is attached for reading if (*shmflg* &SHM_RDONLY) is non-zero and the calling
41456 process has read permission; otherwise, if it is 0 and the calling process has read and write
41457 permission, the segment is attached for reading and writing.

41458 **RETURN VALUE**

41459 Upon successful completion, *shmat*( ) shall increment the value of *shm_nattch* in the data
41460 structure associated with the shared memory ID of the attached shared memory segment and
41461 return the segment's start address.

41462 Otherwise, the shared memory segment shall not be attached, *shmat*( ) shall return –1, and *errno*
41463 shall be set to indicate the error.

41464 **ERRORS**

41465 The *shmat*( ) function shall fail if:

41466 [EACCES] Operation permission is denied to the calling process; see Section 2.7 (on page
41467 489).

41468 [EINVAL] The value of *shmid* is not a valid shared memory identifier, the *shmaddr* is not a
41469 null pointer, and the value of (*shmaddr* –((*uintptr_t*)*shmaddr* %SHMLBA)) is an
41470 illegal address for attaching shared memory; or the *shmaddr* is not a null
41471 pointer, (*shmflg* &SHM_RND) is 0, and the value of *shmaddr* is an illegal
41472 address for attaching shared memory.

41473 [EMFILE] The number of shared memory segments attached to the calling process
41474 would exceed the system-imposed limit.

41475 [ENOMEM] The available data space is not large enough to accommodate the shared
41476 memory segment.

41477 **EXAMPLES**

41478 None.

41479 **APPLICATION USAGE**

41480 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
41481 Application developers who need to use IPC should design their applications so that modules
41482 using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the
41483 alternative interfaces.

41484 **RATIONALE**

41485 None.

41486 **FUTURE DIRECTIONS**

41487 None.

41488 **SEE ALSO**

41489 *exec*, *exit*( ), *fork*( ), *shmctl*( ), *shmdt*( ), *shmget*( ), *shm_open*( ), *shm_unlink*( ), the Base Definitions
41490 volume of IEEE Std 1003.1-200x, **<sys/shm.h>**, Section 2.7 (on page 489)

41491 **CHANGE HISTORY**

41492 First released in Issue 2. Derived from Issue 2 of the SVID.

41493 **Issue 5**

41494 Moved from SHARED MEMORY to BASE.

41495 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
41496 DIRECTIONS to a new APPLICATION USAGE section.

41497 **Issue 6**

41498 The Open Group Corrigendum U021/13 is applied.

41499 **NAME**

41500 shmctl — XSI shared memory control operations

41501 **SYNOPSIS**

41502 XSI
```
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```
41503

41504

41505 **DESCRIPTION**

41506 The *shmctl*( ) function operates on XSI shared memory (see the Base Definitions volume of
41507 IEEE Std 1003.1-200x, Section 3.340, Shared Memory Object). It is unspecified whether this
41508 function interoperates with the realtime interprocess communication facilities defined in Section
41509 2.8 (on page 491).

41510 The *shmctl*( ) function provides a variety of shared memory control operations as specified by
41511 *cmd*. The following values for *cmd* are available:

41512 IPC_STAT — Place the current value of each member of the **shmid_ds** data structure
41513 associated with *shmid* into the structure pointed to by *buf*. The contents of the
41514 structure are defined in **<sys/shm.h>**.

41515 IPC_SET — Set the value of the following members of the **shmid_ds** data structure
41516 associated with *shmid* to the corresponding value found in the structure
41517 pointed to by *buf*:

41518 `shm_perm.uid`
41519 `shm_perm.gid`
41520 `shm_perm.mode` Low-order nine bits.

41521 IPC_SET can only be executed by a process that has an effective user ID equal
41522 to either that of a process with appropriate privileges or to the value of
41523 *shm_perm.cuid* or *shm_perm.uid* in the **shmid_ds** data structure associated with
41524 *shmid*.

41525 IPC_RMID — Remove the shared memory identifier specified by *shmid* from the system and
41526 destroy the shared memory segment and **shmid_ds** data structure associated
41527 with it. IPC_RMID can only be executed by a process that has an effective user
41528 ID equal to either that of a process with appropriate privileges or to the value
41529 of *shm_perm.cuid* or *shm_perm.uid* in the **shmid_ds** data structure associated
41530 with *shmid*.

41531 **RETURN VALUE**

41532 Upon successful completion, *shmctl*( ) shall return 0; otherwise, it shall return –1 and set *errno* to
41533 indicate the error.

41534 **ERRORS**

41535 The *shmctl*( ) function shall fail if:

41536 [EACCES] — The argument *cmd* is equal to IPC_STAT and the calling process does not have
41537 read permission; see Section 2.7 (on page 489).

41538 [EINVAL] — The value of *shmid* is not a valid shared memory identifier, or the value of *cmd*
41539 is not a valid command.

41540 [EPERM] — The argument *cmd* is equal to IPC_RMID or IPC_SET and the effective user ID
41541 of the calling process is not equal to that of a process with appropriate
41542 privileges and it is not equal to the value of *shm_perm.cuid* or *shm_perm.uid* in
41543 the data structure associated with *shmid*.

The *shmctl*( ) function may fail if:

[EOVERFLOW] The *cmd* argument is IPC_STAT and the *gid* or *uid* value is too large to be stored in the structure pointed to by the *buf* argument.

EXAMPLES

None.

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

shmat( ), *shmdt*( ), *shmget*( ), *shm_open*( ), *shm_unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/shm.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

Moved from SHARED MEMORY to BASE.

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

41567 **NAME**

41568 shmdt — XSI shared memory detach operation

41569 **SYNOPSIS**

41570 XSI
```
#include <sys/shm.h>

int shmdt(const void *shmaddr);
```
41571

41572

41573 **DESCRIPTION**

41574 The *shmdt*( ) function operates on XSI shared memory (see the Base Definitions volume of
41575 IEEE Std 1003.1-200x, Section 3.340, Shared Memory Object). It is unspecified whether this
41576 function interoperates with the realtime interprocess communication facilities defined in Section
41577 2.8 (on page 491).

41578 The *shmdt*( ) function detaches the shared memory segment located at the address specified by
41579 *shmaddr* from the address space of the calling process.

41580 **RETURN VALUE**

41581 Upon successful completion, *shmdt*( ) shall decrement the value of *shm_nattch* in the data
41582 structure associated with the shared memory ID of the attached shared memory segment and
41583 return 0.

41584 Otherwise, the shared memory segment shall not be detached, *shmdt*( ) shall return –1, and *errno*
41585 shall be set to indicate the error.

41586 **ERRORS**

41587 The *shmdt*( ) function shall fail if:

41588 [EINVAL] The value of *shmaddr* is not the data segment start address of a shared
41589 memory segment.

41590 **EXAMPLES**

41591 None.

41592 **APPLICATION USAGE**

41593 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
41594 Application developers who need to use IPC should design their applications so that modules
41595 using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the
41596 alternative interfaces.

41597 **RATIONALE**

41598 None.

41599 **FUTURE DIRECTIONS**

41600 None.

41601 **SEE ALSO**

41602 *exec*, *exit*( ), *fork*( ), *shmat*( ), *shmctl*( ), *shmget*( ), *shm_open*( ), *shm_unlink*( ), the Base Definitions
41603 volume of IEEE Std 1003.1-200x, **<sys/shm.h>**, Section 2.7 (on page 489)

41604 **CHANGE HISTORY**

41605 First released in Issue 2. Derived from Issue 2 of the SVID.

41606 **Issue 5**

41607 Moved from SHARED MEMORY to BASE.

41608 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
41609 DIRECTIONS to a new APPLICATION USAGE section.

41610 **NAME**

41611 shmget — get XSI shared memory segment

41612 **SYNOPSIS**

41613 XSI
```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```
41614

41615

41616 **DESCRIPTION**

41617 The *shmget*( ) function operates on XSI shared memory (see the Base Definitions volume of
41618 IEEE Std 1003.1-200x, Section 3.340, Shared Memory Object). It is unspecified whether this
41619 function interoperates with the realtime interprocess communication facilities defined in Section
41620 2.8 (on page 491).

41621 The *shmget*( ) function shall return the shared memory identifier associated with *key*.

41622 A shared memory identifier, associated data structure, and shared memory segment of at least
41623 *size* bytes (see **<sys/shm.h>**) are created for *key* if one of the following is true:

41624 • The argument *key* is equal to IPC_PRIVATE.

41625 • The argument *key* does not already have a shared memory identifier associated with it and
41626 (*shmflg* &IPC_CREAT) is non-zero.

41627 Upon creation, the data structure associated with the new shared memory identifier shall be
41628 initialized as follows:

41629 • The values of *shm_perm.cuid*, *shm_perm.uid*, *shm_perm.cgid*, and *shm_perm.gid* are set equal to
41630 the effective user ID and effective group ID, respectively, of the calling process.

41631 • The low-order nine bits of *shm_perm.mode* are set equal to the low-order nine bits of *shmflg*.
41632 The value of *shm_segsz* is set equal to the value of *size*.

41633 • The values of *shm_lpid*, *shm_nattch*, *shm_atime*, and *shm_dtime* are set equal to 0.

41634 • The value of *shm_ctime* is set equal to the current time.

41635 When the shared memory segment is created, it shall be initialized with all zero values.

41636 **RETURN VALUE**

41637 Upon successful completion, *shmget*( ) shall return a non-negative integer, namely a shared
41638 memory identifier; otherwise, it shall return –1 and set *errno* to indicate the error.

41639 **ERRORS**

41640 The *shmget*( ) function shall fail if:

41641 [EACCES] A shared memory identifier exists for *key* but operation permission as
41642 specified by the low-order nine bits of *shmflg* would not be granted; see
41643 Section 2.7 (on page 489).

41644 [EEXIST] A shared memory identifier exists for the argument *key* but (*shmflg*
41645 &IPC_CREAT) &&(*shmflg* &IPC_EXCL) is non-zero. |

41646 [EINVAL] A shared memory segment is to be created and the value of size is less than |
41647 the system-imposed minimum or greater than the system-imposed maximum. |

41648 [EINVAL] No shared memory segment is to be created and a shared memory segment |
41649 exists for *key* but the size of the segment associated with it is less than *size* and |
41650 *size* is not 0.

41651 [ENOENT] A shared memory identifier does not exist for the argument *key* and (*shmflg*
41652 &IPC_CREAT) is 0.

41653 [ENOMEM] A shared memory identifier and associated shared memory segment shall be
41654 created, but the amount of available physical memory is not sufficient to fill
41655 the request.

41656 [ENOSPC] A shared memory identifier is to be created, but the system-imposed limit on
41657 the maximum number of allowed shared memory identifiers system-wide
41658 would be exceeded.

41659 **EXAMPLES**

41660 None.

41661 **APPLICATION USAGE**

41662 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
41663 Application developers who need to use IPC should design their applications so that modules
41664 using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the
41665 alternative interfaces.

41666 **RATIONALE**

41667 None.

41668 **FUTURE DIRECTIONS**

41669 None.

41670 **SEE ALSO**

41671 *shmat*( ), *shmctl*( ), *shmdt*( ), *shm_open*( ), *shm_unlink*( ), the Base Definitions volume of
41672 IEEE Std 1003.1-200x, **<sys/shm.h>**, Section 2.7 (on page 489)

41673 **CHANGE HISTORY**

41674 First released in Issue 2. Derived from Issue 2 of the SVID.

41675 **Issue 5**

41676 Moved from SHARED MEMORY to BASE.

41677 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
41678 DIRECTIONS to a new APPLICATION USAGE section.

41679 **NAME**

41680 shutdown — shut down socket send and receive operations

41681 **SYNOPSIS**

41682 `#include <sys/socket.h>`

41683 `int shutdown(int socket, int how);`

41684 **DESCRIPTION**

41685 The *shutdown*( ) function shall cause all or part of a full-duplex connection on the socket
41686 associated with the file descriptor *socket* to be shut down.

41687 The *shutdown*( ) function takes the following arguments:

41688 *socket* Specifies the file descriptor of the socket.

41689 *how* Specifies the type of shutdown. The values are as follows:

41690 SHUT_RD Disables further receive operations.

41691 SHUT_WR Disables further send operations.

41692 SHUT_RDWR Disables further send and receive operations.

41693 The *shutdown*( ) function disables subsequent send and/or receive operations on a socket,
41694 depending on the value of the *how* argument.

41695 **RETURN VALUE**

41696 Upon successful completion, *shutdown*( ) shall return 0; otherwise, –1 shall be returned and *errno*
41697 set to indicate the error.

41698 **ERRORS**

41699 The *shutdown*( ) function shall fail if:

41700 [EBADF] The *socket* argument is not a valid file descriptor.

41701 [EINVAL] The *how* argument is invalid.

41702 [ENOTCONN] The socket is not connected.

41703 [ENOTSOCK] The *socket* argument does not refer to a socket.

41704 The *shutdown*( ) function may fail if:

41705 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

41706 **EXAMPLES**

41707 None.

41708 **APPLICATION USAGE**

41709 None.

41710 **RATIONALE**

41711 None.

41712 **FUTURE DIRECTIONS**

41713 None.

41714 **SEE ALSO**

41715 *getsockopt*( ), *read*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *send*( ), *sendto*( ), *setsockopt*( ), *socket*( ),
41716 *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

41717 **CHANGE HISTORY**

41718 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

41719 **NAME**

41720 sigaction — examine and change signal action

41721 **SYNOPSIS**

41722 CX
```
#include <signal.h>

int sigaction(int sig, const struct sigaction *restrict act,
    struct sigaction *restrict oact);
```

41723
41724
41725

41726 **DESCRIPTION**

41727 The *sigaction*( ) function allows the calling process to examine and/or specify the action to be
41728 associated with a specific signal. The argument *sig* specifies the signal; acceptable values are
41729 defined in **<signal.h>**.

41730 The structure **sigaction**, used to describe an action to be taken, is defined in the **<signal.h>**
41731 header to include at least the following members:

41732
41733

Member Type	Member Name	Description
void(*) (int)	*sa_handler*	SIG_DFL, SIG_IGN, or pointer to a function.
sigset_t	*sa_mask*	Additional set of signals to be blocked during execution of signal-catching function.
int	*sa_flags*	Special flags to affect behavior of signal.
void(*) (int, siginfo_t *, void *)	*sa_sigaction*	Signal-catching function.

41734
41735
41736
41737
41738
41739
41740

41741 The storage occupied by *sa_handler* and *sa_sigaction* may overlap, and a conforming application
41742 shall not use both simultaneously.

41743 If the argument *act* is not a null pointer, it points to a structure specifying the action to be
41744 associated with the specified signal. If the argument *oact* is not a null pointer, the action
41745 previously associated with the signal is stored in the location pointed to by the argument *oact*. If
41746 the argument *act* is a null pointer, signal handling is unchanged; thus, the call can be used to
41747 enquire about the current handling of a given signal. The SIGKILL and SIGSTOP signals shall
41748 not be added to the signal mask using this mechanism; this restriction shall be enforced by the
41749 system without causing an error to be indicated.

41750 If the SA_SIGINFO flag (see below) is cleared in the *sa_flags* field of the **sigaction** structure, the
41751 XSI|RTS *sa_handler* field identifies the action to be associated with the specified signal. If the
41752 SA_SIGINFO flag is set in the *sa_flags* field, and the implementation supports the Realtime
41753 Signals Extension option or the X/Open System Interfaces Extension option, the *sa_sigaction*
41754 field specifies a signal-catching function. If the SA_SIGINFO bit is cleared and the *sa_handler*
41755 field specifies a signal-catching function, or if the SA_SIGINFO bit is set, the *sa_mask* field
41756 identifies a set of signals that shall be added to the signal mask of the thread before the signal-
41757 catching function is invoked. If the *sa_handler* field specifies a signal-catching function, the
41758 *sa_mask* field identifies a set of signals that shall be added to the process' signal mask before the
41759 signal-catching function is invoked.

41760 The *sa_flags* field can be used to modify the behavior of the specified signal.

41761 The following flags, defined in the **<signal.h>** header, can be set in *sa_flags*:

41762 XSI SA_NOCLDSTOP Do not generate SIGCHLD when children stop or stopped children
41763 continue.

If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is not set in *sa_flags*, and the implementation supports the SIGCHLD signal, then a SIGCHLD signal shall be generated for the calling process whenever any of its child processes stop and a SIGCHLD signal may be generated for the calling process whnever any of its stopped child processes are continued. If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is set in *sa_flags*, then the implementation shall not generate a SIGCHLD signal in this way.

XSI SA_ONSTACK If set and an alternate signal stack has been declared with *sigaltstack*( ) or *sigstack*( ), the signal shall be delivered to the calling process on that stack. Otherwise, the signal shall be delivered on the current stack.

XSI SA_RESETHAND If set, the disposition of the signal shall be reset to SIG_DFL and the SA_SIGINFO flag shall be cleared on entry to the signal handler.

Note: SIGILL and SIGTRAP cannot be automatically reset when delivered; the system silently enforces this restriction.

Otherwise, the disposition of the signal shall not be modified on entry to the signal handler.

In addition, if this flag is set, *sigaction*( ) behaves as if the SA_NODEFER flag were also set.

XSI SA_RESTART This flag affects the behavior of interruptible functions; that is, those specified to fail with *errno* set to [EINTR]. If set, and a function specified as interruptible is interrupted by this signal, the function shall restart and shall not fail with [EINTR] unless otherwise specified. If the flag is not set, interruptible functions interrupted by this signal shall fail with *errno* set to [EINTR].

SA_SIGINFO If cleared and the signal is caught, the signal-catching function shall be entered as:

```
void func(int signo);
```

where *signo* is the only argument to the signal catching function. In this case, the application shall use the *sa_handler* member to describe the signal catching function and the application shall not modify the *sa_sigaction* member.

XSI|RTS If SA_SIGINFO is set and the signal is caught, the signal-catching function shall be entered as:

```
void func(int signo, siginfo_t *info, void *context);
```

where two additional arguments are passed to the signal catching function. The second argument shall point to an object of type **siginfo_t** explaining the reason why the signal was generated; the third argument can be cast to a pointer to an object of type **ucontext_t** to refer to the receiving process' context that was interrupted when the signal was delivered. In this case, the application shall use the *sa_sigaction* member to describe the signal catching function and the application shall not modify the *sa_handler* member.

The *si_signo* member contains the system-generated signal number.

XSI The *si_errno* member may contain implementation-defined additional error information; if non-zero, it contains an error number identifying the condition that caused the signal to be generated.

41810 XSI|RTS The *si_code* member contains a code identifying the cause of the signal. |

41811 XSI If the value of *si_code* is less than or equal to 0, then the signal was |
41812 generated by a process and *si_pid* and *si_uid*, respectively, indicate the
41813 process ID and the real user ID of the sender. The **<signal.h>** header
41814 description contains information about the signal specific contents of the
41815 elements of the **siginfo_t** type.

41816 XSI SA_NOCLDWAIT If set, and *sig* equals SIGCHLD, child processes of the calling processes
41817 shall not be transformed into zombie processes when they terminate. If
41818 the calling process subsequently waits for its children, and the process
41819 has no unwaited-for children that were transformed into zombie
41820 processes, it shall block until all of its children terminate, and *wait*(),
41821 *waitid*(), and *waitpid*() shall fail and set *errno* to [ECHILD]. Otherwise,
41822 terminating child processes shall be transformed into zombie processes,
41823 unless SIGCHLD is set to SIG_IGN.

41824 XSI SA_NODEFER If set and *sig* is caught, *sig* shall not be added to the process' signal mask
41825 on entry to the signal handler unless it is included in *sa_mask*. Otherwise,
41826 *sig* shall always be added to the process' signal mask on entry to the
41827 signal handler.

41828 When a signal is caught by a signal-catching function installed by *sigaction*(), a new signal mask
41829 is calculated and installed for the duration of the signal-catching function (or until a call to either
41830 *sigprocmask*() or *sigsuspend*() is made). This mask is formed by taking the union of the current
41831 XSI signal mask and the value of the *sa_mask* for the signal being delivered unless SA_NODEFER or
41832 SA_RESETHAND is set, and then including the signal being delivered. If and when the user's
41833 signal handler returns normally, the original signal mask is restored.

41834 Once an action is installed for a specific signal, it shall remain installed until another action is
41835 XSI explicitly requested (by another call to *sigaction*()), until the SA_RESETHAND flag causes
41836 resetting of the handler, or until one of the *exec* functions is called.

41837 If the previous action for *sig* had been established by *signal*(), the values of the fields returned in
41838 the structure pointed to by *oact* are unspecified, and in particular *oact->sa_handler* is not
41839 necessarily the same value passed to *signal*(). However, if a pointer to the same structure or a
41840 copy thereof is passed to a subsequent call to *sigaction*() via the *act* argument, handling of the
41841 signal shall be as if the original call to *signal*() were repeated.

41842 If *sigaction*() fails, no new signal handler is installed.

41843 It is unspecified whether an attempt to set the action for a signal that cannot be caught or
41844 ignored to SIG_DFL is ignored or causes an error to be returned with *errno* set to [EINVAL].

41845 If SA_SIGINFO is not set in *sa_flags*, then the disposition of subsequent occurrences of *sig* when
41846 it is already pending is implementation-defined; the signal-catching function shall be invoked
41847 RTS with a single argument. If the implementation supports the Realtime Signals Extension option,
41848 and if SA_SIGINFO is set in *sa_flags*, then subsequent occurrences of *sig* generated by *sigqueue*()
41849 or as a result of any signal-generating function that supports the specification of an application-
41850 defined value (when *sig* is already pending) shall be queued in FIFO order until delivered or
41851 accepted; the signal-catching function shall be invoked with three arguments. The application
41852 specified value is passed to the signal-catching function as the *si_value* member of the **siginfo_t**
41853 structure.

41854 The result of the use of *sigaction*() and a *sigwait*() function concurrently within a process on the
41855 same signal is unspecified.

RETURN VALUE

Upon successful completion, *sigaction*( ) shall return 0; otherwise, –1 shall be returned, *errno* shall be set to indicate the error, and no new signal-catching function shall be installed.

ERRORS

The *sigaction*( ) function shall fail if:

[EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

[ENOTSUP] The SA_SIGINFO bit flag is set in the *sa_flags* field of the **sigaction** structure, and the implementation does not support either the Realtime Signals Extension option, or the X/Open System Interfaces Extension option.

The *sigaction*( ) function may fail if:

[EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be caught or ignored (or both).

EXAMPLES

None.

APPLICATION USAGE

The *sigaction*( ) function supersedes the *signal*( ) function, and should be used in preference. In particular, *sigaction*( ) and *signal*( ) should not be used in the same process to control the same signal. The behavior of reentrant functions, as defined in the DESCRIPTION, is as specified by this volume of IEEE Std 1003.1-200x, regardless of invocation from a signal-catching function. This is the only intended meaning of the statement that reentrant functions may be used in signal-catching functions without restrictions. Applications must still consider all effects of such functions on such things as data structures, files, and process state. In particular, application writers need to consider the restrictions on interactions when interrupting *sleep*( ) and interactions among multiple handles for a file description. The fact that any specific function is listed as reentrant does not necessarily mean that invocation of that function from a signal-catching function is recommended.

In order to prevent errors arising from interrupting non-reentrant function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some programmatic semaphore (see *semget*( ), *sem_init*( ), *sem_open*( ), and so on). Note in particular that even the ‘‘safe’’ functions may modify *errno*; the signal-catching function, if not executing as an independent thread, may want to save and restore its value. Naturally, the same principles apply to the reentrancy of application routines and asynchronous data access. Note that *longjmp*( ) and *siglongjmp*( ) are not in the list of reentrant functions. This is because the code executing after *longjmp*( ) and *siglongjmp*( ) can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use *longjmp*( ) and *siglongjmp*( ) from within signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either *malloc*( ) or *free*( ) functions or the standard I/O library, both of which traditionally use data structures in a non-reentrant manner. Since any combination of different functions using a common data structure can cause reentrancy problems, this volume of IEEE Std 1003.1-200x does not define the behavior when any unsafe function is called in a signal handler that interrupts an unsafe function.

If the signal occurs other than as the result of calling *abort*( ), *kill*( ), or *raise*( ), the behavior is undefined if the signal handler calls any function in the standard library other than one of the functions listed in the table above or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type **volatile sig_atomic_t**. Furthermore, if such a call fails, the value of *errno* is unspecified.

Usually, the signal is executed on the stack that was in effect before the signal was delivered. An alternate stack may be specified to receive a subset of the signals being caught.

When the signal handler returns, the receiving process resumes execution at the point it was interrupted unless the signal handler makes other arrangements. If *longjmp*( ) or *_longjmp*( ) is used to leave the signal handler, then the signal mask must be explicitly restored by the process.

This volume of IEEE Std 1003.1-200x defines the third argument of a signal handling function when SA_SIGINFO is set as a **void *** instead of a **ucontext_t ***, but without requiring type checking. New applications should explicitly cast the third argument of the signal handling function to **ucontext_t ***.

The BSD optional four argument signal handling function is not supported by this volume of IEEE Std 1003.1-200x. The BSD declaration would be:

```
void handler(int sig, int code, struct sigcontext *scp,
    char *addr);
```

where *sig* is the signal number, *code* is additional information on certain signals, *scp* is a pointer to the sigcontext structure, and *addr* is additional address information. Much the same information is available in the objects pointed to by the second argument of the signal handler specified when SA_SIGINFO is set.

RATIONALE

Although this volume of IEEE Std 1003.1-200x requires that signals that cannot be ignored shall not be added to the signal mask when a signal-catching function is entered, there is no explicit requirement that subsequent calls to *sigaction*( ) reflect this in the information returned in the *oact* argument. In other words, if SIGKILL is included in the *sa_mask* field of *act*, it is unspecified whether or not a subsequent call to *sigaction*( ) returns with SIGKILL included in the *sa_mask* field of *oact*.

The SA_NOCLDSTOP flag, when supplied in the *act->sa_flags* parameter, allows overloading SIGCHLD with the System V semantics that each SIGCLD signal indicates a single terminated child. Most conforming applications that catch SIGCHLD are expected to install signal-catching functions that repeatedly call the *waitpid*( ) function with the WNOHANG flag set, acting on each child for which status is returned, until *waitpid*( ) returns zero. If stopped children are not of interest, the use of the SA_NOCLDSTOP flag can prevent the overhead from invoking the signal-catching routine when they stop.

Some historical implementations also define other mechanisms for stopping processes, such as the *ptrace*( ) function. These implementations usually do not generate a SIGCHLD signal when processes stop due to this mechanism; however, that is beyond the scope of this volume of IEEE Std 1003.1-200x.

This volume of IEEE Std 1003.1-200x requires that calls to *sigaction*( ) that supply a NULL *act* argument succeed, even in the case of signals that cannot be caught or ignored (that is, SIGKILL or SIGSTOP). The System V *signal*( ) and BSD *sigvec*( ) functions return [EINVAL] in these cases and, in this respect, their behavior varies from *sigaction*( ).

This volume of IEEE Std 1003.1-200x requires that *sigaction*( ) properly save and restore a signal action set up by the ISO C standard *signal*( ) function. However, there is no guarantee that the reverse is true, nor could there be given the greater amount of information conveyed by the **sigaction** structure. Because of this, applications should avoid using both functions for the same signal in the same process. Since this cannot always be avoided in case of general-purpose library routines, they should always be implemented with *sigaction*( ).

It was intended that the *signal*( ) function should be implementable as a library routine using *sigaction*( ).

The POSIX Realtime Extension extends the *sigaction*( ) function as specified by the POSIX.1-1990 standard to allow the application to request on a per-signal basis via an additional signal action flag that the extra parameters, including the application-defined signal value, if any, be passed to the signal-catching function.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 478), *bsd_signal*( ), *kill*( ), *_longjmp*( ), *longjmp*( ), *raise*( ), *semget*( ), *sem_init*( ), *sem_open*( ), *sigaddset*( ), *sigaltstack*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *signal*( ), *sigprocmask*( ), *sigsuspend*( ), *wait*( ), *waitid*( ), *waitpid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<ucontext.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and POSIX Threads Extension.

In the DESCRIPTION, the second argument to *func* when SA_SIGINFO is set is no longer permitted to be NULL, and the description of permitted **siginfo_t** contents is expanded by reference to **<signal.h>**.

Since the X/OPEN UNIX Extension functionality is now folded into the BASE, the [ENOTSUP] error is deleted.

Issue 6

The Open Group Corrigendum U028/7 is applied. In the paragraph entitled ‘‘Signal Effects on Other Functions’’, a reference to *sigpending*( ) is added.

In the DESCRIPTION, the text ‘‘Signal Generation and Delivery’’, ‘‘Signal Actions’’, and ‘‘Signal Effects on Other Functions’’ are moved to a separate section of this volume of IEEE Std 1003.1-200x.

Text describing functionality from the Realtime Signals option is marked.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENOTSUP] error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *sigaction*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

References to the *wait3*( ) function are removed.

The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an extension over the ISO C standard.

41988 **NAME**

41989 sigaddset — add a signal to a signal set

41990 **SYNOPSIS**

41991 CX
```
#include <signal.h>

int sigaddset(sigset_t *set, int signo);
```
41992

41993

41994 **DESCRIPTION**

41995 The *sigaddset*( ) function adds the individual signal specified by the *signo* to the signal set pointed
41996 to by *set*.

41997 Applications shall call either *sigemptyset*( ) or *sigfillset*( ) at least once for each object of type
41998 **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is
41999 nonetheless supplied as an argument to any of *pthread_sigmask*( ), *sigaction*( ), *sigaddset*( ),
42000 *sigdelset*( ), *sigismember*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), *sigtimedwait*( ), *sigwait*( ), or
42001 *sigwaitinfo*( ), the results are undefined.

42002 **RETURN VALUE**

42003 Upon successful completion, *sigaddset*( ) shall return 0; otherwise, it shall return –1 and set *errno*
42004 to indicate the error.

42005 **ERRORS**

42006 The *sigaddset*( ) function may fail if:

42007 [EINVAL] The value of the *signo* argument is an invalid or unsupported signal number.

42008 **EXAMPLES**

42009 None.

42010 **APPLICATION USAGE**

42011 None.

42012 **RATIONALE**

42013 None.

42014 **FUTURE DIRECTIONS**

42015 None.

42016 **SEE ALSO**

42017 Section 2.4 (on page 478), *sigaction*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ),
42018 *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
42019 **<signal.h>**

42020 **CHANGE HISTORY**

42021 First released in Issue 3.

42022 Entry included for alignment with the POSIX.1-1988 standard.

42023 **Issue 5**

42024 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
42025 previous issues.

42026 **Issue 6**

42027 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

42028 The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an
42029 extension over the ISO C standard.

42030 **NAME**

42031 sigaltstack — set and get signal alternate stack context

42032 **SYNOPSIS**

42033 XSI
```
#include <signal.h>

int sigaltstack(const stack_t *restrict ss, stack_t *restrict oss);
```
42034

42035

42036 **DESCRIPTION**

42037 The *sigaltstack*( ) function allows a process to define and examine the state of an alternate stack
42038 for signal handlers. Signals that have been explicitly declared to execute on the alternate stack
42039 shall be delivered on the alternate stack.

42040 If *ss* is not a null pointer, it points to a **stack_t** structure that specifies the alternate signal stack
42041 that shall take effect upon return from *sigaltstack*( ). The *ss_flags* member specifies the new stack
42042 state. If it is set to SS_DISABLE, the stack is disabled and *ss_sp* and *ss_size* are ignored.
42043 Otherwise, the stack shall be enabled, and the *ss_sp* and *ss_size* members specify the new address
42044 and size of the stack.

42045 The range of addresses starting at *ss_sp* up to but not including *ss_sp*+*ss_size*, is available to the
42046 implementation for use as the stack. This function makes no assumptions regarding which end
42047 is the stack base and in which direction the stack grows as items are pushed.

42048 If *oss* is not a null pointer, on successful completion it shall point to a **stack_t** structure that
42049 specifies the alternate signal stack that was in effect prior to the call to *sigaltstack*( ). The *ss_sp*
42050 and *ss_size* members specify the address and size of that stack. The *ss_flags* member specifies the
42051 stack's state, and may contain one of the following values:

42052 SS_ONSTACK The process is currently executing on the alternate signal stack. Attempts to
42053 modify the alternate signal stack while the process is executing on it fail. This
42054 flag shall not be modified by processes.

42055 SS_DISABLE The alternate signal stack is currently disabled.

42056 The value SIGSTKSZ is a system default specifying the number of bytes that would be used to
42057 cover the usual case when manually allocating an alternate stack area. The value MINSIGSTKSZ
42058 is defined to be the minimum stack size for a signal handler. In computing an alternate stack
42059 size, a program should add that amount to its stack requirements to allow for the system
42060 implementation overhead. The constants SS_ONSTACK, SS_DISABLE, SIGSTKSZ, and
42061 MINSIGSTKSZ are defined in **<signal.h>**.

42062 After a successful call to one of the *exec* functions, there are no alternate signal stacks in the new
42063 process image.

42064 In some implementations, a signal (whether or not indicated to execute on the alternate stack)
42065 shall always execute on the alternate stack if it is delivered while another signal is being caught
42066 using the alternate stack.

42067 Use of this function by library threads that are not bound to kernel-scheduled entities results in
42068 undefined behavior.

42069 **RETURN VALUE**

42070 Upon successful completion, *sigaltstack*( ) shall return 0; otherwise, it shall return –1 and set *errno*
42071 to indicate the error.

42072 **ERRORS**

42073 The *sigaltstack*( ) function shall fail if:

42074 [EINVAL] The *ss* argument is not a null pointer, and the *ss_flags* member pointed to by *ss*
42075 contains flags other than SS_DISABLE.

42076 [ENOMEM] The size of the alternate stack area is less than MINSIGSTKSZ.

42077 [EPERM] An attempt was made to modify an active stack.

42078 **EXAMPLES**

42079 **Allocating Memory for an Alternate Stack**

42080 The following example illustrates a method for allocating memory for an alternate stack.

```
42081 #include <signal.h>
42082 ...
42083 if ((sigstk.ss_sp = malloc(SIGSTKSZ)) == NULL)
42084     /* Error return. */
42085 sigstk.ss_size = SIGSTKSZ;
42086 sigstk.ss_flags = 0;
42087 if (sigaltstack(&sigstk,(stack_t *)0) < 0)
42088     perror("sigaltstack");
```

42089 **APPLICATION USAGE**

42090 On some implementations, stack space is automatically extended as needed. On those
42091 implementations, automatic extension is typically not available for an alternate stack. If the stack
42092 overflows, the behavior is undefined.

42093 **RATIONALE**

42094 None.

42095 **FUTURE DIRECTIONS**

42096 None.

42097 **SEE ALSO**

42098 Section 2.4 (on page 478), *sigaction*( ), *sigsetjmp*( ), the Base Definitions volume of
42099 IEEE Std 1003.1-200x, **<signal.h>**

42100 **CHANGE HISTORY**

42101 First released in Issue 4, Version 2.

42102 **Issue 5**

42103 Moved from X/OPEN UNIX extension to BASE.

42104 The last sentence of the DESCRIPTION was included as an APPLICATION USAGE note in
42105 previous issues.

42106 **Issue 6**

42107 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

42108 The **restrict** keyword is added to the *sigaltstack*( ) prototype for alignment with the
42109 ISO/IEC 9899: 1999 standard.

42110 **NAME**

42111 sigdelset — delete a signal from a signal set

42112 **SYNOPSIS**

42113 CX
```
#include <signal.h>

int sigdelset(sigset_t *set, int signo);
```
42114

42115

42116 **DESCRIPTION**

42117 The *sigdelset*( ) function deletes the individual signal specified by *signo* from the signal set
42118 pointed to by *set*.

42119 Applications should call either *sigemptyset*( ) or *sigfillset*( ) at least once for each object of type
42120 **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is
42121 nonetheless supplied as an argument to any of *pthread_sigmask*( ), *sigaction*( ), *sigaddset*( ),
42122 *sigdelset*( ), *sigismember*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), *sigtimedwait*( ), *sigwait*( ), or
42123 *sigwaitinfo*( ), the results are undefined.

42124 **RETURN VALUE**

42125 Upon successful completion, *sigdelset*( ) shall return 0; otherwise, it shall return –1 and set *errno*
42126 to indicate the error.

42127 **ERRORS**

42128 The *sigdelset*( ) function may fail if:

42129 [EINVAL] The *signo* argument is not a valid signal number, or is an unsupported signal
42130 number.

42131 **EXAMPLES**

42132 None.

42133 **APPLICATION USAGE**

42134 None.

42135 **RATIONALE**

42136 None.

42137 **FUTURE DIRECTIONS**

42138 None.

42139 **SEE ALSO**

42140 Section 2.4 (on page 478), *sigaction*( ), *sigaddset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ),
42141 *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
42142 **<signal.h>**

42143 **CHANGE HISTORY**

42144 First released in Issue 3.

42145 Entry included for alignment with the POSIX.1-1988 standard.

42146 **Issue 5**

42147 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
42148 previous issues.

42149 **Issue 6**

42150 The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an
42151 extension over the ISO C standard.

NAME

sigemptyset — initialize and empty a signal set

SYNOPSIS

CX
```
#include <signal.h>

int sigemptyset(sigset_t *set);
```

DESCRIPTION

The *sigemptyset*( ) function initializes the signal set pointed to by *set*, such that all signals defined in IEEE Std 1003.1-200x are excluded.

RETURN VALUE

Upon successful completion, *sigemptyset*( ) shall return 0; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The implementation of the *sigemptyset*( ) (or *sigfillset*( )) function could quite trivially clear (or set) all the bits in the signal set. Alternatively, it would be reasonable to initialize part of the structure, such as a version field, to permit binary-compatibility between releases where the size of the set varies. For such reasons, either *sigemptyset*( ) or *sigfillset*( ) must be called prior to any other use of the signal set, even if such use is read-only (for example, as an argument to *sigpending*( )). This function is not intended for dynamic allocation.

The *sigfillset*( ) and *sigemptyset*( ) functions require that the resulting signal set include (or exclude) all the signals defined in this volume of IEEE Std 1003.1-200x. Although it is outside the scope of this volume of IEEE Std 1003.1-200x to place this requirement on signals that are implemented as extensions, it is recommended that implementation-defined signals also be affected by these functions. However, there may be a good reason for a particular signal not to be affected. For example, blocking or ignoring an implementation-defined signal may have undesirable side effects, whereas the default action for that signal is harmless. In such a case, it would be preferable for such a signal to be excluded from the signal set returned by *sigfillset*( ).

In early proposals there was no distinction between invalid and unsupported signals (the names of optional signals that were not supported by an implementation were not defined by that implementation). The [EINVAL] error was thus specified as a required error for invalid signals. With that distinction, it is not necessary to require implementations of these functions to determine whether an optional signal is actually supported, as that could have a significant performance impact for little value. The error could have been required for invalid signals and optional for unsupported signals, but this seemed unnecessarily complex. Thus, the error is optional in both cases.

FUTURE DIRECTIONS

None.

42195 **SEE ALSO**

42196 Section 2.4 (on page 478), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigfillset*( ), *sigismember*( ),
42197 *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
42198 **<signal.h>**

42199 **CHANGE HISTORY**

42200 First released in Issue 3.

42201 Entry included for alignment with the POSIX.1-1988 standard. |

42202 **Issue 6** |

42203 The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an |
42204 extension over the ISO C standard. |

42205 **NAME**

42206 sigfillset — initialize and fill a signal set

42207 **SYNOPSIS**

42208 CX
```
#include <signal.h>

int sigfillset(sigset_t *set);
```
42209

42210

42211 **DESCRIPTION**

42212 The *sigfillset*( ) function shall initialize the signal set pointed to by *set*, such that all signals
42213 defined in this volume of IEEE Std 1003.1-200x are included.

42214 **RETURN VALUE**

42215 Upon successful completion, *sigfillset*( ) shall return 0; otherwise, it shall return –1 and set *errno*
42216 to indicate the error.

42217 **ERRORS**

42218 No errors are defined.

42219 **EXAMPLES**

42220 None.

42221 **APPLICATION USAGE**

42222 None.

42223 **RATIONALE**

42224 Refer to *sigemptyset*( ) (on page 1848).

42225 **FUTURE DIRECTIONS**

42226 None.

42227 **SEE ALSO**

42228 Section 2.4 (on page 478), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigismember*( ),
42229 *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
42230 **<signal.h>**

42231 **CHANGE HISTORY**

42232 First released in Issue 3.

42233 Entry included for alignment with the POSIX.1-1988 standard.

42234 **Issue 6**

42235 The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an
42236 extension over the ISO C standard.

NAME

sighold, sigignore, sigpause, sigrelse, sigset — signal management

SYNOPSIS

XSI

```
#include <signal.h>

int sighold(int sig);
int sigignore(int sig);
int sigpause(int sig);
int sigrelse(int sig);
void (*sigset(int sig, void (*disp)(int)))(int);
```

DESCRIPTION

Use of any of these functions is unspecified in a multi-threaded process.

The *sighold*( ), *sigignore*( ), *sigpause*( ), *sigrelse*( ), and *sigset*( ) functions provide simplified signal management.

The *sigset*( ) function shall modify signal dispositions. The *sig* argument specifies the signal, which may be any signal except SIGKILL and SIGSTOP. The *disp* argument specifies the signal's disposition, which may be SIG_DFL, SIG_IGN, or the address of a signal handler. If *sigset*( ) is used, and *disp* is the address of a signal handler, the system shall add *sig* to the calling process' signal mask before executing the signal handler; when the signal handler returns, the system shall restore the calling process' signal mask to its state prior to the delivery of the signal. In addition, if *sigset*( ) is used, and *disp* is equal to SIG_HOLD, *sig* shall be added to the calling process' signal mask and *sig*'s disposition shall remain unchanged. If *sigset*( ) is used, and *disp* is not equal to SIG_HOLD, *sig* shall be removed from the calling process' signal mask.

The *sighold*( ) function shall add *sig* to the calling process' signal mask.

The *sigrelse*( ) function shall remove *sig* from the calling process' signal mask.

The *sigignore*( ) function shall set the disposition of *sig* to SIG_IGN.

The *sigpause*( ) function shall remove *sig* from the calling process' signal mask and suspend the calling process until a signal is received. The *sigpause*( ) function shall restore the process' signal mask to its original state before returning.

If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the calling processes shall not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited-for children that were transformed into zombie processes, it shall block until all of its children terminate, and *wait*( ), *waitid*( ), and *waitpid*( ) shall fail and set *errno* to [ECHILD].

RETURN VALUE

Upon successful completion, *sigset*( ) shall return SIG_HOLD if the signal had been blocked and the signal's previous disposition if it had not been blocked. Otherwise, SIG_ERR shall be returned and *errno* set to indicate the error.

The *sigpause*( ) function shall suspend execution of the thread until a signal is received, whereupon it shall return –1 and set *errno* to [EINTR].

For all other functions, upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to indicate the error.

42279 **ERRORS**

42280 These functions shall fail if:

42281 [EINVAL] The *sig* argument is an illegal signal number.

42282 The *sigset*( ) and *sigignore*( ) functions shall fail if:

42283 [EINVAL] An attempt is made to catch a signal that cannot be caught, or to ignore a
42284 signal that cannot be ignored.

42285 **EXAMPLES**

42286 None.

42287 **APPLICATION USAGE**

42288 The *sigaction*( ) function provides a more comprehensive and reliable mechanism for controlling
42289 signals; new applications should use *sigaction*( ) rather than *sigset*( ).

42290 The *sighold*( ) function, in conjunction with *sigrelse*( ) or *sigpause*( ), may be used to establish
42291 critical regions of code that require the delivery of a signal to be temporarily deferred.

42292 The *sigsuspend*( ) function should be used in preference to *sigpause*( ) for broader portability.

42293 **RATIONALE**

42294 None.

42295 **FUTURE DIRECTIONS**

42296 None.

42297 **SEE ALSO**

42298 Section 2.4 (on page 478), *exec*, *pause*( ), *sigaction*( ), *signal*( ), *sigsuspend*( ), *waitid*( ), the Base
42299 Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

42300 **CHANGE HISTORY**

42301 First released in Issue 4 Version 2.

42302 **Issue 5**

42303 Moved from X/OPEN UNIX extension to BASE.

42304 The DESCRIPTION is updated to indicate that the *sigpause*( ) function restores the process'
42305 signal mask to its original state before returning.

42306 The RETURN VALUE section is updated to indicate that the *sigpause*( ) function suspends
42307 execution of the process until a signal is received, whereupon it returns –1 and sets *errno* to
42308 [EINTR].

42309 **Issue 6**

42310 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

42311 References to the *wait3*( ) function are removed.

42312 The XSI functions are split out into their own reference page.

42313 **NAME**

42314 sigignore — signal management

42315 **SYNOPSIS**

42316 XSI
```
#include <signal.h>

int sigignore(int sig);
```
42317

42318

42319 **DESCRIPTION**

42320 Refer to *sighold*( ).

42321 **NAME**

42322 siginterrupt — allow signals to interrupt functions

42323 **SYNOPSIS**

42324 XSI
```
#include <signal.h>
```

42325
```
int siginterrupt(int sig, int flag);
```

42326

42327 **DESCRIPTION**

42328 The *siginterrupt*( ) function shall change the restart behavior when a function is interrupted by |
42329 the specified signal. The function *siginterrupt*(*sig*, *flag*) has an effect as if implemented as:

42330
```
siginterrupt(int sig, int flag) {
    int ret;
    struct sigaction act;

    (void) sigaction(sig, NULL, &act);
    if (flag)
        act.sa_flags &= ~SA_RESTART;
    else
        act.sa_flags |= SA_RESTART;
    ret = sigaction(sig, &act, NULL);
    return ret;
}
```

42341 **RETURN VALUE**

42342 Upon successful completion, *siginterrupt*( ) shall return 0; otherwise, –1 shall be returned and
42343 *errno* set to indicate the error.

42344 **ERRORS**

42345 The *siginterrupt*( ) function shall fail if:

42346 [EINVAL] The *sig* argument is not a valid signal number.

42347 **EXAMPLES**

42348 None.

42349 **APPLICATION USAGE**

42350 The *siginterrupt*( ) function supports programs written to historical system interfaces. A |
42351 conforming application, when being written or rewritten, should use *sigaction*( ) with the |
42352 SA_RESTART flag instead of *siginterrupt*( ).

42353 **RATIONALE**

42354 None.

42355 **FUTURE DIRECTIONS**

42356 None.

42357 **SEE ALSO**

42358 Section 2.4 (on page 478), *sigaction*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
42359 **<signal.h>**

42360 **CHANGE HISTORY**

42361 First released in Issue 4, Version 2.

42362 **Issue 5**
42363 Moved from X/OPEN UNIX extension to BASE.

42364 **NAME**

42365 sigismember — test for a signal in a signal set

42366 **SYNOPSIS**

42367 CX
```
#include <signal.h>

int sigismember(const sigset_t *set, int signo);
```
42368

42369

42370 **DESCRIPTION**

42371 The *sigismember*( ) function shall test whether the signal specified by *signo* is a member of the set
42372 pointed to by *set*.

42373 Applications should call either *sigemptyset*( ) or *sigfillset*( ) at least once for each object of type
42374 **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is
42375 nonetheless supplied as an argument to any of *pthread_sigmask*( ), *sigaction*( ), *sigaddset*( ),
42376 *sigdelset*( ), *sigismember*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), *sigtimedwait*( ), *sigwait*( ), or
42377 *sigwaitinfo*( ), the results are undefined.

42378 **RETURN VALUE**

42379 Upon successful completion, *sigismember*( ) shall return 1 if the specified signal is a member of
42380 the specified set, or 0 if it is not. Otherwise, it shall return –1 and set *errno* to indicate the error.

42381 **ERRORS**

42382 The *sigismember*( ) function may fail if:

42383 [EINVAL] The *signo* argument is not a valid signal number, or is an unsupported signal
42384 number.

42385 **EXAMPLES**

42386 None.

42387 **APPLICATION USAGE**

42388 None.

42389 **RATIONALE**

42390 None.

42391 **FUTURE DIRECTIONS**

42392 None.

42393 **SEE ALSO**

42394 Section 2.4 (on page 478), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigfillset*( ), *sigemptyset*( ),
42395 *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
42396 **<signal.h>**

42397 **CHANGE HISTORY**

42398 First released in Issue 3.

42399 Entry included for alignment with the POSIX.1-1988 standard.

42400 **Issue 5**

42401 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
42402 previous issues.

42403 **Issue 6**

42404 The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an
42405 extension over the ISO C standard.

42406 **NAME**

42407 siglongjmp — non-local goto with signal handling

42408 **SYNOPSIS**

42409 CX
```
#include <setjmp.h>

void siglongjmp(sigjmp_buf env, int val);
```
42410

42411

42412 **DESCRIPTION**

42413 The *siglongjmp*( ) function shall be equivalent to the *longjmp*( ) function, except as follows:

42414 • References to *setjmp*( ) shall be equivalent to *sigsetjmp*( ).

42415 • The *siglongjmp*( ) function shall restore the saved signal mask if and only if the *env* argument
42416 was initialized by a call to *sigsetjmp*( ) with a non-zero *savemask* argument.

42417 **RETURN VALUE**

42418 After *siglongjmp*( ) is completed, program execution shall continue as if the corresponding
42419 invocation of *sigsetjmp*( ) had just returned the value specified by *val*. The *siglongjmp*( ) function
42420 shall not cause *sigsetjmp*( ) to return 0; if *val* is 0, *sigsetjmp*( ) shall return the value 1.

42421 **ERRORS**

42422 No errors are defined.

42423 **EXAMPLES**

42424 None.

42425 **APPLICATION USAGE**

42426 The distinction between *setjmp*( ) or *longjmp*( ) and *sigsetjmp*( ) or *siglongjmp*( ) is only significant
42427 for programs which use *sigaction*( ), *sigprocmask*( ), or *sigsuspend*( ).

42428 **RATIONALE**

42429 None.

42430 **FUTURE DIRECTIONS**

42431 None.

42432 **SEE ALSO**

42433 *longjmp*( ), *setjmp*( ), *sigprocmask*( ), *sigsetjmp*( ), *sigsuspend*( ), the Base Definitions volume of
42434 IEEE Std 1003.1-200x, **<setjmp.h>**

42435 **CHANGE HISTORY**

42436 First released in Issue 3.

42437 Entry included for alignment with the ISO POSIX-1 standard.

42438 **Issue 5**

42439 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

42440 **Issue 6**

42441 The DESCRIPTION is rewritten in terms of *longjmp*( ).

42442 The SYNOPSIS is marked CX since the presence of this function in the **<setjmp.h>** header is an
42443 extension over the ISO C standard.

42444 **NAME**

42445 signal — signal management

42446 **SYNOPSIS**

42447
```
#include <signal.h>
```

42448
```
void (*signal(int sig, void (*func)(int)))(int);
```

42449 **DESCRIPTION**

42450 CX The functionality described on this reference page is aligned with the ISO C standard. Any
42451 conflict between the requirements described here and the ISO C standard is unintentional. This
42452 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

42453 CX Use of this function is unspecified in a multi-threaded process.

42454 The *signal*( ) function chooses one of three ways in which receipt of the signal number *sig* is to be
42455 subsequently handled. If the value of *func* is SIG_DFL, default handling for that signal shall
42456 occur. If the value of *func* is SIG_IGN, the signal shall be ignored. Otherwise, the application
42457 shall ensure that *func* points to a function to be called when that signal occurs. An invocation of
42458 such a function because of a signal, or (recursively) of any further functions called by that
42459 invocation (other than functions in the standard library), is called a ''signal handler''.

42460 When a signal occurs, and *func* points to a function, it is implementation-defined whether the
42461 equivalent of a:

42462
```
signal(sig, SIG_DFL);
```

42463 is executed or the implementation prevents some implementation-defined set of signals (at least
42464 including *sig*) from occurring until the current signal handling has completed. (If the value of *sig*
42465 is SIGILL, the implementation may alternatively define that no action is taken.) Next the
42466 equivalent of:

42467
```
(*func)(sig);
```

42468 is executed. If and when the function returns, if the value of *sig* was SIGFPE, SIGILL, or
42469 SIGSEGV or any other implementation-defined value corresponding to a computational
42470 exception, the behavior is undefined. Otherwise, the program shall resume execution at the
42471 CX point it was interrupted. If the signal occurs as the result of calling the *abort*( ), *raise*( ), *kill*( ), |
42472 *pthread_kill*( ), or *sigqueue*( ) function, the signal handler shall not call the *raise*( ) function. |

42473 CX If the signal occurs other than as the result of calling *abort*( ), *raise*( ), *kill*( ), *pthread_kill*( ), or |
42474 *sigqueue*( ), the behavior is undefined if the signal handler refers to any object with static storage |
42475 duration other than by assigning a value to an object declared as volatile **sig_atomic_t**, or if the
42476 signal handler calls any function in the standard library other than one of the functions listed in
42477 Section 2.4 (on page 478). Furthermore, if such a call fails, the value of *errno* is unspecified. |

42478 At program start-up, the equivalent of:

42479
```
signal(sig, SIG_IGN);
```

42480 is executed for some signals, and the equivalent of:

42481
```
signal(sig, SIG_DFL);
```

42482 CX is executed for all other signals (see *exec*).

42483 **RETURN VALUE**

42484 If the request can be honored, *signal*( ) shall return the value of *func* for the most recent call to
42485 *signal*( ) for the specified signal *sig*. Otherwise, SIG_ERR shall be returned and a positive value
42486 shall be stored in *errno*.

42487 **ERRORS**

42488 The *signal*( ) function shall fail if:

42489 CX [EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a
42490 signal that cannot be caught or ignore a signal that cannot be ignored.

42491 The *signal*( ) function may fail if:

42492 CX [EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be
42493 caught or ignored (or both).

42494 **EXAMPLES**

42495 None.

42496 **APPLICATION USAGE**

42497 The *sigaction*( ) function provides a more comprehensive and reliable mechanism for controlling
42498 signals; new applications should use *sigaction*( ) rather than *signal*( ).

42499 **RATIONALE**

42500 None.

42501 **FUTURE DIRECTIONS**

42502 None.

42503 **SEE ALSO**

42504 Section 2.4 (on page 478), *exec*, *pause*( ), *sigaction*( ), *sigsuspend*( ), *waitid*( ), the Base Definitions
42505 volume of IEEE Std 1003.1-200x, **<signal.h>**

42506 **CHANGE HISTORY**

42507 First released in Issue 1. Derived from Issue 1 of the SVID.

42508 **Issue 5**

42509 Moved from X/OPEN UNIX extension to BASE.

42510 The DESCRIPTION is updated to indicate that the *sigpause*( ) function restores the process'
42511 signal mask to its original state before returning.

42512 The RETURN VALUE section is updated to indicate that the *sigpause*( ) function suspends
42513 execution of the process until a signal is received, whereupon it returns –1 and sets *errno* to
42514 [EINTR].

42515 **Issue 6**

42516 Extensions beyond the ISO C standard are now marked.

42517 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

42518 The DESCRIPTION is updated for alignment with the ISO/IEC 9899: 1999 standard.

42519 References to the *wait3*( ) function are removed.

42520 The *sighold*( ), *sigignore*( ), *sigrelse*( ), and *sigset*( ) functions are split out onto their own reference
42521 page.

42522 **NAME**

42523 signbit — test sign

42524 **SYNOPSIS**

42525
```
#include <math.h>
```

42526
```
int signbit(real-floating x);
```

42527 **DESCRIPTION**

42528 CX The functionality described on this reference page is aligned with the ISO C standard. Any
42529 conflict between the requirements described here and the ISO C standard is unintentional. This
42530 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

42531 The *signbit*( ) macro shall determine whether the sign of its argument value is negative. NaNs,
42532 zeros, and infinities have a sign bit.

42533 **RETURN VALUE**

42534 The *signbit*( ) macro shall return a non-zero value if and only if the sign of its argument value is
42535 negative.

42536 **ERRORS**

42537 No errors are defined.

42538 **EXAMPLES**

42539 None.

42540 **APPLICATION USAGE**

42541 None.

42542 **RATIONALE**

42543 None.

42544 **FUTURE DIRECTIONS**

42545 None.

42546 **SEE ALSO**

42547 *fpclassify*( ), *isfinite*( ), *isinf*( ), *isnan*( ), *isnormal*( ), the Base Definitions volume of
42548 IEEE Std 1003.1-200x, **<math.h>**

42549 **CHANGE HISTORY**

42550 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

42551 **NAME**

42552 sigpause — remove a signal from the signal mask and suspend the thread

42553 **SYNOPSIS**

42554 XSI
```
#include <signal.h>
```

42555
```
int sigpause(int sig);
```

42556

42557 **DESCRIPTION**

42558 Refer to *sighold*( ).

42559 **NAME**

42560 sigpending — examine pending signals

42561 **SYNOPSIS**

42562 CX
```
#include <signal.h>

int sigpending(sigset_t *set);
```
42563

42564

42565 **DESCRIPTION**

42566 The *sigpending*( ) function shall store, in the location referenced by the *set* argument, the set of
42567 signals that are blocked from delivery to the calling thread and that are pending on the process
42568 or the calling thread.

42569 **RETURN VALUE**

42570 Upon successful completion, *sigpending*( ) shall return 0; otherwise, –1 shall be returned and
42571 *errno* set to indicate the error.

42572 **ERRORS**

42573 No errors are defined.

42574 **EXAMPLES**

42575 None.

42576 **APPLICATION USAGE**

42577 None.

42578 **RATIONALE**

42579 None.

42580 **FUTURE DIRECTIONS**

42581 None.

42582 **SEE ALSO**

42583 *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *sigprocmask*( ), the Base Definitions
42584 volume of IEEE Std 1003.1-200x, **<signal.h>**

42585 **CHANGE HISTORY**

42586 First released in Issue 3.

42587 **Issue 5**

42588 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

42589 **Issue 6**

42590 The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an
42591 extension over the ISO C standard.

42592 **NAME**

42593 sigprocmask — examine and change blocked signals

42594 **SYNOPSIS**

42595 CX
```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set,
    sigset_t *restrict oset);
```
42598

42599 **DESCRIPTION**

42600 Refer to *pthread_sigmask*( ).

42601 **NAME**

42602 sigqueue — queue a signal to a process (**REALTIME**)

42603 **SYNOPSIS**

42604 RTS
```
#include <signal.h>

int sigqueue(pid_t pid, int signo, const union sigval value);
```
42606

42607 **DESCRIPTION**

42608 The *sigqueue*( ) function shall cause the signal specified by *signo* to be sent with the value
42609 specified by *value* to the process specified by *pid*. If *signo* is zero (the null signal), error checking
42610 is performed but no signal is actually sent. The null signal can be used to check the validity of
42611 *pid*.

42612 The conditions required for a process to have permission to queue a signal to another process
42613 are the same as for the *kill*( ) function.

42614 The *sigqueue*( ) function shall return immediately. If SA_SIGINFO is set for *signo* and if the
42615 resources were available to queue the signal, the signal shall be queued and sent to the receiving
42616 process. If SA_SIGINFO is not set for *signo*, then *signo* shall be sent at least once to the receiving
42617 process; it is unspecified whether *value* shall be sent to the receiving process as a result of this
42618 call.

42619 If the value of *pid* causes *signo* to be generated for the sending process, and if *signo* is not blocked
42620 for the calling thread and if no other thread has *signo* unblocked or is waiting in a *sigwait*( )
42621 function for *signo*, either *signo* or at least the pending, unblocked signal shall be delivered to the
42622 calling thread before the *sigqueue*( ) function returns. Should any multiple pending signals in the
42623 range SIGRTMIN to SIGRTMAX be selected for delivery, it shall be the lowest numbered one.
42624 The selection order between realtime and non-realtime signals, or between multiple pending
42625 non-realtime signals, is unspecified.

42626 **RETURN VALUE**

42627 Upon successful completion, the specified signal shall have been queued, and the *sigqueue*( )
42628 function shall return a value of zero. Otherwise, the function shall return a value of –1 and set
42629 *errno* to indicate the error.

42630 **ERRORS**

42631 The *sigqueue*( ) function shall fail if:

42632 [EAGAIN] No resources available to queue the signal. The process has already queued
42633 SIGQUEUE_MAX signals that are still pending at the receiver(s), or a system-
42634 wide resource limit has been exceeded.

42635 [EINVAL] The value of the *signo* argument is an invalid or unsupported signal number.

42636 [EPERM] The process does not have the appropriate privilege to send the signal to the
42637 receiving process.

42638 [ESRCH] The process *pid* does not exist.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *sigqueue*( ) function allows an application to queue a realtime signal to itself or to another process, specifying the application-defined value. This is common practice in realtime applications on existing realtime systems. It was felt that specifying another function in the *sig*... name space already carved out for signals was preferable to extending the interface to *kill*( ).

Such a function became necessary when the put/get event function of the message queues was removed. It should be noted that the *sigqueue*( ) function implies reduced performance in a security-conscious implementation as the access permissions between the sender and receiver have to be checked on each send when the *pid* is resolved into a target process. Such access checks were necessary only at message queue open in the previous interface.

The standard developers required that *sigqueue*( ) have the same semantics with respect to the null signal as *kill*( ), and that the same permission checking be used. But because of the difficulty of implementing the ''broadcast'' semantic of *kill*( ) (for example, to process groups) and the interaction with resource allocation, this semantic was not adopted. The *sigqueue*( ) function queues a signal to a single process specified by the *pid* argument.

The *sigqueue*( ) function can fail if the system has insufficient resources to queue the signal. An explicit limit on the number of queued signals that a process could send was introduced. While the limit is ''per-sender'', this volume of IEEE Std 1003.1-200x does not specify that the resources be part of the state of the sender. This would require either that the sender be maintained after exit until all signals that it had sent to other processes were handled or that all such signals that had not yet been acted upon be removed from the queue(s) of the receivers. This volume of IEEE Std 1003.1-200x does not preclude this behavior, but an implementation that allocated queuing resources from a system-wide pool (with per-sender limits) and that leaves queued signals pending after the sender exits is also permitted.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.8.1 (on page 491), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6

The *sigqueue*( ) function is marked as part of the Realtime Signals Extension option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Realtime Signals Extension option.

42679 **NAME**

42680 sigrelse — remove a signal from signal mask or modify signal disposition

42681 **SYNOPSIS**

42682 XSI
```
#include <signal.h>
```

42683
```
int sigrelse(int sig);
```

42684

42685 **DESCRIPTION**

42686 Refer to *sighold*( ).

42687 **NAME**

42688 sigset — signal management

42689 **SYNOPSIS**

42690 `#include <signal.h>`

42691 XSI `void (*sigset(int sig, void (*disp)(int)))(int);`

42692

42693 **DESCRIPTION**

42694 Refer to *sighold*( ).

42695 **NAME**

42696 sigsetjmp — set jump point for a non-local goto

42697 **SYNOPSIS**

42698 CX
```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
```
42699

42700

42701 **DESCRIPTION**

42702 The *sigsetjmp*( ) function shall be equivalent to the *setjmp*( ) function, except as follows:

42703 • References to *setjmp*( ) are equivalent to *sigsetjmp*( ).

42704 • References to *longjmp*( ) are equivalent to *siglongjmp*( ).

42705 • If the value of the *savemask* argument is not 0, *sigsetjmp*( ) shall also save the current signal
42706 mask of the calling thread as part of the calling environment.

42707 **RETURN VALUE**

42708 If the return is from a successful direct invocation, *sigsetjmp*( ) shall return 0. If the return is from
42709 a call to *siglongjmp*( ), *sigsetjmp*( ) shall return a non-zero value.

42710 **ERRORS**

42711 No errors are defined.

42712 **EXAMPLES**

42713 None.

42714 **APPLICATION USAGE**

42715 The distinction between *setjmp*( )/*longjmp*( ) and *sigsetjmp*( )/*siglongjmp*( ) is only significant for
42716 programs which use *sigaction*( ), *sigprocmask*( ), or *sigsuspend*( ).

42717 Note that since this function is defined in terms of *setjmp*( ), if *savemask* is zero, it is unspecified
42718 whether the signal mask is saved.

42719 **RATIONALE**

42720 The ISO C standard specifies various restrictions on the usage of the *setjmp*( ) macro in order to
42721 permit implementors to recognize the name in the compiler and not implement an actual
42722 function. These same restrictions apply to the *sigsetjmp*( ) macro.

42723 There are processors that cannot easily support these calls, but this was not considered a
42724 sufficient reason to exclude them.

42725 4.2 BSD, 4.3 BSD, and XSI-conformant systems provide functions named *_setjmp*( ) and
42726 *_longjmp*( ) that, together with *setjmp*( ) and *longjmp*( ), provide the same functionality as
42727 *sigsetjmp*( ) and *siglongjmp*( ). On those systems, *setjmp*( ) and *longjmp*( ) save and restore signal
42728 masks, while *_setjmp*( ) and *_longjmp*( ) do not. On System V, Release 3 and in corresponding
42729 issues of the SVID, *setjmp*( ) and *longjmp*( ) are explicitly defined not to save and restore signal
42730 masks. In order to permit existing practice in both cases, the relation of *setjmp*( ) and *longjmp*( ) to
42731 signal masks is not specified, and a new set of functions is defined instead.

42732 The *longjmp*( ) and *siglongjmp*( ) functions operate as in the previous issue provided the matching
42733 *setjmp*( ) or *sigsetjmp*( ) has been performed in the same thread. Non-local jumps into contexts
42734 saved by other threads would be at best a questionable practice and were not considered worthy
42735 of standardization.

FUTURE DIRECTIONS

None.

SEE ALSO

siglongjmp(), *signal*(), *sigprocmask*(), *sigsuspend*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<setjmp.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The DESCRIPTION is reworded in terms of *setjmp*(). |

The SYNOPSIS is marked CX since the presence of this function in the **<setjmp.h>** header is an extension over the ISO C standard. | |

42750 **NAME**

42751 sigsuspend — wait for a signal

42752 **SYNOPSIS**

42753 CX
```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```
42754

42755

42756 **DESCRIPTION**

42757 The *sigsuspend*( ) function shall replace the current signal mask of the calling thread with the set
42758 of signals pointed to by *sigmask* and then suspend the thread until delivery of a signal whose
42759 action is either to execute a signal-catching function or to terminate the process. This shall not
42760 cause any other signals that may have been pending on the process to become pending on the
42761 thread.

42762 If the action is to terminate the process then *sigsuspend*( ) shall never return. If the action is to
42763 execute a signal-catching function, then *sigsuspend*( ) shall return after the signal-catching
42764 function returns, with the signal mask restored to the set that existed prior to the *sigsuspend*( )
42765 call.

42766 It is not possible to block signals that cannot be ignored. This is enforced by the system without
42767 causing an error to be indicated.

42768 **RETURN VALUE**

42769 Since *sigsuspend*( ) suspends thread execution indefinitely, there is no successful completion
42770 return value. If a return occurs, –1 shall be returned and *errno* set to indicate the error.

42771 **ERRORS**

42772 The *sigsuspend*( ) function shall fail if:

42773 [EINTR] A signal is caught by the calling process and control is returned from the
42774 signal-catching function.

42775 **EXAMPLES**

42776 None.

42777 **APPLICATION USAGE**

42778 Normally, at the beginning of a critical code section, a specified set of signals is blocked using
42779 the *sigprocmask*( ) function. When the thread has completed the critical section and needs to wait
42780 for the previously blocked signal(s), it pauses by calling *sigsuspend*( ) with the mask that was
42781 returned by the *sigprocmask*( ) call.

42782 **RATIONALE**

42783 None.

42784 **FUTURE DIRECTIONS**

42785 None.

42786 **SEE ALSO**

42787 Section 2.4 (on page 478), *pause*( ), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), the
42788 Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

42789 **CHANGE HISTORY**

42790 First released in Issue 3.

42791 Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The text in the RETURN VALUE section has been changed from ‘‘suspends process execution’’ to ‘‘suspends thread execution’’. This reflects IEEE PASC Interpretation 1003.1c #40.

Text in the APPLICATION USAGE section has been replaced. |

The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an extension over the ISO C standard. | |

42800 **NAME**

42801 sigtimedwait, sigwaitinfo — wait for queued signals (**REALTIME**)

42802 **SYNOPSIS**

42803 RTS
```
#include <signal.h>

int sigtimedwait(const sigset_t *restrict set,
    siginfo_t *restrict info,
    const struct timespec *restrict timeout);
int sigwaitinfo(const sigset_t *restrict set,
    siginfo_t *restrict info);
```
42809

42810 **DESCRIPTION**

42811 The *sigtimedwait*( ) function shall be equivalent to *sigwaitinfo*( ) except that if none of the signals |
42812 specified by *set* are pending, *sigtimedwait*( ) shall wait for the time interval specified in the
42813 **timespec** structure referenced by *timeout*. If the **timespec** structure pointed to by *timeout* is
42814 zero-valued and if none of the signals specified by *set* are pending, then *sigtimedwait*( ) shall
42815 MON return immediately with an error. If *timeout* is the NULL pointer, the behavior is unspecified. If
42816 the Monotonic Clock option is supported, the CLOCK_MONOTONIC clock shall be used to
42817 measure the time interval specified by the *timeout* argument.

42818 The *sigwaitinfo*( ) function selects the pending signal from the set specified by *set*. Should any of
42819 multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it shall be the
42820 lowest numbered one. The selection order between realtime and non-realtime signals, or
42821 between multiple pending non-realtime signals, is unspecified. If no signal in *set* is pending at
42822 the time of the call, the calling thread shall be suspended until one or more signals in *set* become
42823 pending or until it is interrupted by an unblocked, caught signal.

42824 The *sigwaitinfo*( ) function shall be equivalent to the *sigwait*( ) function if the *info* argument is |
42825 NULL. If the *info* argument is non-NULL, the *sigwaitinfo*( ) function shall be equivalent to |
42826 *sigwait*( ), except that the selected signal number shall be stored in the *si_signo* member, and the
42827 cause of the signal shall be stored in the *si_code* member. If any value is queued to the selected
42828 signal, the first such queued value shall be dequeued and, if the *info* argument is non-NULL, the
42829 value shall be stored in the *si_value* member of *info*. The system resource used to queue the
42830 signal shall be released and returned to the system for other use. If no value is queued, the
42831 content of the *si_value* member is undefined. If no further signals are queued for the selected
42832 signal, the pending indication for that signal shall be reset.

42833 **RETURN VALUE**

42834 Upon successful completion (that is, one of the signals specified by *set* is pending or is
42835 generated) *sigwaitinfo*( ) and *sigtimedwait*( ) shall return the selected signal number. Otherwise,
42836 the function shall return a value of –1 and set *errno* to indicate the error.

42837 **ERRORS**

42838 The *sigtimedwait*( ) function shall fail if:

42839 [EAGAIN] No signal specified by *set* was generated within the specified timeout period.

42840 The *sigtimedwait*( ) and *sigwaitinfo*( ) functions may fail if:

42841 [EINTR] The wait was interrupted by an unblocked, caught signal. It shall be
42842 documented in system documentation whether this error causes these
42843 functions to fail.

42844 The *sigtimedwait*( ) function may also fail if:

[EINVAL] The *timeout* argument specified a *tv_nsec* value less than zero or greater than or equal to 1 000 million.

An implementation only checks for this error if no signal is pending in *set* and it is necessary to wait.

EXAMPLES

None.

APPLICATION USAGE

The *sigtimedwait*() function times out and returns an [EAGAIN] error. Application writers should note that this is inconsistent with other functions such as *pthread_cond_timedwait*() that return [ETIMEDOUT].

RATIONALE

Existing programming practice on realtime systems uses the ability to pause waiting for a selected set of events and handle the first event that occurs in-line instead of in a signal-handling function. This allows applications to be written in an event-directed style similar to a state machine. This style of programming is useful for largescale transaction processing in which the overall throughput of an application and the ability to clearly track states are more important than the ability to minimize the response time of individual event handling.

It is possible to construct a signal-waiting macro function out of the realtime signal function mechanism defined in this volume of IEEE Std 1003.1-200x. However, such a macro has to include the definition of a generalized handler for all signals to be waited on. A significant portion of the overhead of handler processing can be avoided if the signal-waiting function is provided by the kernel. This volume of IEEE Std 1003.1-200x therefore provides two signal-waiting functions—one that waits indefinitely and one with a timeout—as part of the overall realtime signal function specification.

The specification of a function with a timeout allows an application to be written that can be broken out of a wait after a set period of time if no event has occurred. It was argued that setting a timer event before the wait and recognizing the timer event in the wait would also implement the same functionality, but at a lower performance level. Because of the performance degradation associated with the user-level specification of a timer event and the subsequent cancelation of that timer event after the wait completes for a valid event, and the complexity associated with handling potential race conditions associated with the user-level method, the separate function has been included.

Note that the semantics of the *sigwaitinfo*() function are nearly identical to that of the *sigwait*() function defined by this volume of IEEE Std 1003.1-200x. The only difference is that *sigwaitinfo*() returns the queued signal value in the *value* argument. The return of the queued value is required so that applications can differentiate between multiple events queued to the same signal number.

The two distinct functions are being maintained because some implementations may choose to implement the POSIX Threads Extension functions and not implement the queued signals extensions. Note, though, that *sigwaitinfo*() does not return the queued value if the *value* argument is NULL, so the POSIX Threads Extension *sigwait*() function can be implemented as a macro on *sigwaitinfo*().

The *sigtimedwait*() function was separated from the *sigwaitinfo*() function to address concerns regarding the overloading of the *timeout* pointer to indicate indefinite wait (no timeout), timed wait, and immediate return, and concerns regarding consistency with other functions where the conditional and timed waits were separate functions from the pure blocking function. The semantics of *sigtimedwait*() are specified such that *sigwaitinfo*() could be implemented as a macro with a NULL pointer for *timeout*.

The *sigwait* functions provide a synchronous mechanism for threads to wait for asynchronously generated signals. One important question was how many threads that are suspended in a call to a *sigwait*( ) function for a signal should return from the call when the signal is sent. Four choices were considered:

1. Return an error for multiple simultaneous calls to *sigwait* functions for the same signal.
2. One or more threads return.
3. All waiting threads return.
4. Exactly one thread returns.

Prohibiting multiple calls to *sigwait*( ) for the same signal was felt to be overly restrictive. The ''one or more'' behavior made implementation of conforming packages easy at the expense of forcing POSIX threads clients to protect against multiple simultaneous calls to *sigwait*( ) in application code in order to achieve predictable behavior. There was concern that the ''all waiting threads'' behavior would result in ''signal broadcast storms'', consuming excessive CPU resources by replicating the signals in the general case. Furthermore, no convincing examples could be presented that delivery to all was either simpler or more powerful than delivery to one.

Thus, the consensus was that exactly one thread that was suspended in a call to a *sigwait* function for a signal should return when that signal occurs. This is not an onerous restriction as:

- A multi-way signal wait can be built from the single-way wait.
- Signals should only be handled by application-level code, as library routines cannot guess what the application wants to do with signals generated for the entire process.
- Applications can thus arrange for a single thread to wait for any given signal and call any needed routines upon its arrival.

In an application that is using signals for interprocess communication, signal processing is typically done in one place. Alternatively, if the signal is being caught so that process cleanup can be done, the signal handler thread can call separate process cleanup routines for each portion of the application. Since the application main line started each portion of the application, it is at the right abstraction level to tell each portion of the application to clean up.

Certainly, there exist programming styles where it is logical to consider waiting for a single signal in multiple threads. A simple *sigwait_multiple*( ) routine can be constructed to achieve this goal. A possible implementation would be to have each *sigwait_multiple*( ) caller registered as having expressed interest in a set of signals. The caller then waits on a thread-specific condition variable. A single server thread calls a *sigwait*( ) function on the union of all registered signals. When the *sigwait*( ) function returns, the appropriate state is set and condition variables are broadcast. New *sigwait_multiple*( ) callers may cause the pending *sigwait*( ) call to be canceled and reissued in order to update the set of signals being waited for.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.8.1 (on page 491), *pause*( ), *pthread_sigmask*( ), *sigaction*( ), *sigpending*( ), *sigsuspend*( ), *sigwait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6

These functions are marked as part of the Realtime Signals Extension option.

The Open Group Corrigendum U035/3 is applied. The SYNOPSIS of the *sigwaitinfo*() function has been corrected so that the second argument is of type **siginfo_t ***.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Realtime Signals Extension option.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the CLOCK_MONOTONIC clock, if supported, is used to measure timeout intervals.

The **restrict** keyword is added to the *sigtimedwait*() and *sigwaitinfo*() prototypes for alignment with the ISO/IEC 9899:1999 standard.

42946 **NAME**

42947 sigwait — wait for queued signals

42948 **SYNOPSIS**

42949 CX
```
#include <signal.h>

int sigwait(const sigset_t *restrict set, int *restrict sig);
```
42950

42951

42952 **DESCRIPTION**

42953 The *sigwait*( ) function shall select a pending signal from *set*, atomically clear it from the system's
42954 set of pending signals, and return that signal number in the location referenced by *sig*. If prior to
42955 the call to *sigwait*( ) there are multiple pending instances of a single signal number, it is
42956 implementation-defined whether upon successful return there are any remaining pending
42957 RTS signals for that signal number. If the implementation supports queued signals and there are
42958 multiple signals queued for the signal number selected, the first such queued signal shall cause a
42959 return from *sigwait*( ) and the remainder shall remain queued. If no signal in *set* is pending at the
42960 time of the call, the thread shall be suspended until one or more becomes pending. The signals
42961 defined by *set* shall have been blocked at the time of the call to *sigwait*( ); otherwise, the behavior
42962 is undefined. The effect of *sigwait*( ) on the signal actions for the signals in *set* is unspecified.

42963 If more than one thread is using *sigwait*( ) to wait for the same signal, no more than one of these
42964 threads shall return from *sigwait*( ) with the signal number. Which thread returns from *sigwait*( )
42965 if more than a single thread is waiting is unspecified.

42966 RTS Should any of the multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it
42967 shall be the lowest numbered one. The selection order between realtime and non-realtime
42968 signals, or between multiple pending non-realtime signals, is unspecified.

42969 **RETURN VALUE**

42970 Upon successful completion, *sigwait*( ) shall store the signal number of the received signal at the
42971 location referenced by *sig* and return zero. Otherwise, an error number shall be returned to
42972 indicate the error.

42973 **ERRORS**

42974 The *sigwait*( ) function may fail if:

42975 [EINVAL] The *set* argument contains an invalid or unsupported signal number.

42976 **EXAMPLES**

42977 None.

42978 **APPLICATION USAGE**

42979 None.

42980 **RATIONALE**

42981 To provide a convenient way for a thread to wait for a signal, this volume of
42982 IEEE Std 1003.1-200x provides the *sigwait*( ) function. For most cases where a thread has to wait
42983 for a signal, the *sigwait*( ) function should be quite convenient, efficient, and adequate.

42984 However, requests were made for a lower-level primitive than *sigwait*( ) and for semaphores that
42985 could be used by threads. After some consideration, threads were allowed to use semaphores
42986 and *sem_post*( ) was defined to be async-signal and async-cancel-safe.

42987 In summary, when it is necessary for code run in response to an asynchronous signal to notify a
42988 thread, *sigwait*( ) should be used to handle the signal. Alternatively, if the implementation
42989 provides semaphores, they also can be used, either following *sigwait*( ) or from within a signal
42990 handling routine previously registered with *sigaction*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 478), Section 2.8.1 (on page 491), *pause*( ), *pthread_sigmask*( ), *sigaction*( ), *sigpending*( ), *sigsuspend*( ), *sigwaitinfo*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6

The RATIONALE section is added.

The **restrict** keyword is added to the *sigwait*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

sigwaitinfo — wait for queued signals (**REALTIME**)

SYNOPSIS

RTS
```
#include <signal.h>

int sigwaitinfo(const sigset_t *restrict set, siginfo_t *restrict info);
```

DESCRIPTION

Refer to *sigtimedwait*( ).

43012 **NAME**

43013 sin, sinf, sinl — sine function

43014 **SYNOPSIS**

43015
```
#include <math.h>
```

43016
```
double sin(double x);
```
43017
```
float sinf(float x);
```
43018
```
long double sinl(long double x);
```

43019 **DESCRIPTION**

43020 CX The functionality described on this reference page is aligned with the ISO C standard. Any
43021 conflict between the requirements described here and the ISO C standard is unintentional. This
43022 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

43023 These functions shall compute the sine of their argument *x*, measured in radians.

43024 An application wishing to check for error situations should set *errno* to zero and call
43025 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
43026 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
43027 zero, an error has occurred.

43028 **RETURN VALUE**

43029 Upon successful completion, these functions shall return the sine of *x*.

43030 MX If *x* is NaN, a NaN shall be returned.

43031 If *x* is ±0, *x* shall be returned.

43032 If *x* is subnormal, a range error may occur and *x* should be returned.

43033 If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-
43034 defined value shall be returned.

43035 **ERRORS**

43036 These functions shall fail if:

43037 MX Domain Error — The *x* argument is ±Inf.

43038 43039 43040 43041 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

43042 These functions may fail if:

43043 MX Range Error — The value of *x* is subnormal

43044 43045 43046 43047 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

Taking the Sine of a 45-Degree Angle

```
#include <math.h>
...
double radians = 45.0 * M_PI / 180;
double result;
...
result = sin(radians);
```

APPLICATION USAGE

These functions may lose accuracy when their argument is near a multiple of π or is far from 0.0.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asin( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes in previous issues.

Issue 6

The *sinf*( ) and *sinl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

43078 **NAME**

43079 sinf — sine function

43080 **SYNOPSIS**

43081 `#include <math.h>`

43082 `float sinf(float x);`

43083 **DESCRIPTION**

43084 Refer to *sin*( ).

43085 **NAME**

43086 sinh, sinhf, sinhl — hyperbolic sine function

43087 **SYNOPSIS**

43088
```
#include <math.h>
```

43089
```
double sinh(double x);
```
43090
```
float sinhf(float x);
```
43091
```
long double sinhl(long double x);
```

43092 **DESCRIPTION**

43093 CX The functionality described on this reference page is aligned with the ISO C standard. Any
43094 conflict between the requirements described here and the ISO C standard is unintentional. This
43095 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

43096 These functions shall compute the hyperbolic sine of their argument *x*.

43097 An application wishing to check for error situations should set *errno* to zero and call
43098 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
43099 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
43100 zero, an error has occurred.

43101 **RETURN VALUE**

43102 Upon successful completion, these functions shall return the hyperbolic sine of *x*.

43103 If the result would cause an overflow, a range error shall occur and ±HUGE_VAL,
43104 ±HUGE_VALF, and ±HUGE_VALL (with the same sign as *x*) shall be returned as appropriate for
43105 the type of the function.

43106 MX If *x* is NaN, a NaN shall be returned.

43107 If *x* is ±0, or ±Inf, *x* shall be returned.

43108 If *x* is subnormal, a range error may occur and *x* should be returned.

43109 **ERRORS**

43110 These functions shall fail if:

43111 Range Error The result would cause an overflow.

43112 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
43113 then *errno* shall be set to [ERANGE]. If the integer expression |
43114 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow |
43115 floating-point exception shall be raised. |

43116 These functions may fail if:

43117 MX Range Error The value *x* is subnormal.

43118 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
43119 then *errno* shall be set to [ERANGE]. If the integer expression |
43120 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow |
43121 floating-point exception shall be raised. |

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asinh( ), *cosh*( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *tanh*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *sinhf*( ) and *sinhl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

43146 **NAME**

43147 sinl — sine function

43148 **SYNOPSIS**

43149 `#include <math.h>`

43150 `long double sinl(long double x);`

43151 **DESCRIPTION**

43152 Refer to *sin*( ).

43153 **NAME**

43154 sleep — suspend execution for an interval of time

43155 **SYNOPSIS**

43156 `#include <unistd.h>`

43157 `unsigned sleep(unsigned seconds);`

43158 **DESCRIPTION**

43159 The *sleep*( ) function shall cause the calling thread to be suspended from execution until either
43160 the number of realtime seconds specified by the argument *seconds* has elapsed or a signal is
43161 delivered to the calling thread and its action is to invoke a signal-catching function or to
43162 terminate the process. The suspension time may be longer than requested due to the scheduling
43163 of other activity by the system.

43164 If a SIGALRM signal is generated for the calling process during execution of *sleep*( ) and if the
43165 SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether *sleep*( )
43166 returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also
43167 unspecified whether it remains pending after *sleep*( ) returns or it is discarded.

43168 If a SIGALRM signal is generated for the calling process during execution of *sleep*( ), except as a
43169 result of a prior call to *alarm*( ), and if the SIGALRM signal is not being ignored or blocked from
43170 delivery, it is unspecified whether that signal has any effect other than causing *sleep*( ) to return.

43171 If a signal-catching function interrupts *sleep*( ) and examines or changes either the time a
43172 SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or
43173 whether the SIGALRM signal is blocked from delivery, the results are unspecified.

43174 If a signal-catching function interrupts *sleep*( ) and calls *siglongjmp*( ) or *longjmp*( ) to restore an
43175 environment saved prior to the *sleep*( ) call, the action associated with the SIGALRM signal and
43176 the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also
43177 unspecified whether the SIGALRM signal is blocked, unless the process' signal mask is restored
43178 as part of the environment.

43179 XSI Interactions between *sleep*( ) and any of *setitimer*( ), *ualarm*( ), or *usleep*( ) are unspecified.

43180 **RETURN VALUE**

43181 If *sleep*( ) returns because the requested time has elapsed, the value returned shall be 0. If *sleep*( )
43182 returns due to delivery of a signal, the return value shall be the ''unslept'' amount (the requested
43183 time minus the time actually slept) in seconds.

43184 **ERRORS**

43185 No errors are defined.

43186 **EXAMPLES**

43187 None.

43188 **APPLICATION USAGE**

43189 None.

43190 **RATIONALE**

43191 There are two general approaches to the implementation of the *sleep*( ) function. One is to use the
43192 *alarm*( ) function to schedule a SIGALRM signal and then suspend the process waiting for that
43193 signal. The other is to implement an independent facility. This volume of IEEE Std 1003.1-200x
43194 permits either approach.

43195 In order to comply with the requirement that no primitive shall change a process attribute unless
43196 explicitly described by this volume of IEEE Std 1003.1-200x, an implementation using SIGALRM
43197 must carefully take into account any SIGALRM signal scheduled by previous *alarm*( ) calls, the

action previously established for SIGALRM, and whether SIGALRM was blocked. If a SIGALRM has been scheduled before the *sleep*( ) would ordinarily complete, the *sleep*( ) must be shortened to that time and a SIGALRM generated (possibly simulated by direct invocation of the signal-catching function) before *sleep*( ) returns. If a SIGALRM has been scheduled after the *sleep*( ) would ordinarily complete, it must be rescheduled for the same time before *sleep*( ) returns. The action and blocking for SIGALRM must be saved and restored.

Historical implementations often implement the SIGALRM-based version using *alarm*( ) and *pause*( ). One such implementation is prone to infinite hangups, as described in *pause*( ). Another such implementation uses the C-language *setjmp*( ) and *longjmp*( ) functions to avoid that window. That implementation introduces a different problem: when the SIGALRM signal interrupts a signal-catching function installed by the user to catch a different signal, the *longjmp*( ) aborts that signal-catching function. An implementation based on *sigprocmask*( ), *alarm*( ), and *sigsuspend*( ) can avoid these problems.

Despite all reasonable care, there are several very subtle, but detectable and unavoidable, differences between the two types of implementations. These are the cases mentioned in this volume of IEEE Std 1003.1-200x where some other activity relating to SIGALRM takes place, and the results are stated to be unspecified. All of these cases are sufficiently unusual as not to be of concern to most applications.

See also the discussion of the term *realtime* in *alarm*( ).

Since *sleep*( ) can be implemented using *alarm*( ), the discussion about alarms occurring early under *alarm*( ) applies to *sleep*( ) as well. |

Application writers should note that the type of the argument *seconds* and the return value of *sleep*( ) is **unsigned**. That means that a Strictly Conforming POSIX System Interfaces Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX}, which the ISO C standard sets as 65 535, and any application passing a larger value is restricting its portability. A different type was considered, but historical implementations, including those with a 16-bit **int** type, consistently use either **unsigned** or **int**.

Scheduling delays may cause the process to return from the *sleep*( ) function significantly after the requested time. In such cases, the return value should be set to zero, since the formula (requested time minus the time actually spent) yields a negative number and *sleep*( ) returns an **unsigned**.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *getitimer*( ), *nanosleep*( ), *pause*( ), *sigaction*( ), *sigsetjmp*( ), *ualarm*( ), *usleep*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

43238 **NAME**

43239 snprintf — print formatted output

43240 **SYNOPSIS**

43241 `#include <stdio.h>`

43242 `int snprintf(char *restrict s, size_t n,`
43243 `    const char *restrict format, ...);`

43244 **DESCRIPTION**

43245 Refer to *fprintf*( ).

43246 **NAME**

43247 socket — create an endpoint for communication

43248 **SYNOPSIS**

43249 `#include <sys/socket.h>`

43250 `int socket(int domain, int type, int protocol);`

43251 **DESCRIPTION**

43252 The *socket*( ) function shall create an unbound socket in a communications domain, and return a
43253 file descriptor that can be used in later function calls that operate on sockets.

43254 The *socket*( ) function takes the following arguments:

43255 *domain* Specifies the communications domain in which a socket is to be created.

43256 *type* Specifies the type of socket to be created.

43257 *protocol* Specifies a particular protocol to be used with the socket. Specifying a *protocol*
43258 of 0 causes *socket*( ) to use an unspecified default protocol appropriate for the
43259 requested socket type.

43260 The *domain* argument specifies the address family used in the communications domain. The
43261 address families supported by the system are implementation-defined.

43262 Symbolic constants that can be used for the domain argument are defined in the **<sys/socket.h>**
43263 header.

43264 The *type* argument specifies the socket type, which determines the semantics of communication
43265 over the socket. The following socket types are defined; implementations may specify additional
43266 socket types:

43267 SOCK_STREAM Provides sequenced, reliable, bidirectional, connection-mode byte
43268 streams, and may provide a transmission mechanism for out-of-band
43269 data.

43270 SOCK_DGRAM Provides datagrams, which are connectionless-mode, unreliable messages
43271 of fixed maximum length.

43272 SOCK_SEQPACKET Provides sequenced, reliable, bidirectional, connection-mode
43273 transmission path for records. A record can be sent using one or more
43274 output operations and received using one or more input operations, but a
43275 single operation never transfers part of more than one record. Record
43276 boundaries are visible to the receiver via the MSG_EOR flag.

43277 If the *protocol* argument is non-zero, it shall specify a protocol that is supported by the address
43278 family. If the *protocol* argument is zero, the default protocol for this address family and type shall
43279 be used. The protocols supported by the system are implementation-defined.

43280 The process may need to have appropriate privileges to use the *socket*( ) function or to create
43281 some sockets.

43282 **RETURN VALUE**

43283 Upon successful completion, *socket*( ) shall return a non-negative integer, the socket file
43284 descriptor. Otherwise, a value of –1 shall be returned and *errno* set to indicate the error.

43285 **ERRORS**

43286 The *socket*( ) function shall fail if:

43287 [EAFNOSUPPORT]
43288 The implementation does not support the specified address family.

[EMFILE] No more file descriptors are available for this process.

[ENFILE] No more file descriptors are available for the system.

[EPROTONOSUPPORT]
The protocol is not supported by the address family, or the protocol is not supported by the implementation.

[EPROTOTYPE] The socket type is not supported by the protocol.

The *socket*( ) function may fail if:

[EACCES] The process does not have appropriate privileges.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

EXAMPLES

None.

APPLICATION USAGE

The documentation for specific address families specifies which protocols each address family supports. The documentation for specific protocols specifies which socket types each protocol supports.

The application can determine whether an address family is supported by trying to create a socket with *domain* set to the protocol in question.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

accept( ), *bind*( ), *connect*( ), *getsockname*( ), *getsockopt*( ), *listen*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *send*( ), *sendmsg*( ), *setsockopt*( ), *shutdown*( ), *socketpair*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<netinet/in.h>**, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

43317 **NAME**

43318 socketpair — create a pair of connected sockets

43319 **SYNOPSIS**

43320 `#include <sys/socket.h>`

43321 `int socketpair(int domain, int type, int protocol,`
43322 `    int socket_vector[2]);`

43323 **DESCRIPTION**

43324 The *socketpair*( ) function shall create an unbound pair of connected sockets in a specified *domain*,
43325 of a specified *type*, under the protocol optionally specified by the *protocol* argument. The two
43326 sockets shall be identical. The file descriptors used in referencing the created sockets shall be
43327 returned in *socket_vector*[0] and *socket_vector*[1].

43328 The *socketpair*( ) function takes the following arguments:

43329 *domain* Specifies the communications domain in which the sockets are to be created.

43330 *type* Specifies the type of sockets to be created.

43331 *protocol* Specifies a particular protocol to be used with the sockets. Specifying a
43332 *protocol* of 0 causes *socketpair*( ) to use an unspecified default protocol
43333 appropriate for the requested socket type.

43334 *socket_vector* Specifies a 2-integer array to hold the file descriptors of the created socket
43335 pair.

43336 The *type* argument specifies the socket type, which determines the semantics of communications
43337 over the socket. The following socket types are defined; implementations may specify additional
43338 socket types:

43339 SOCK_STREAM Provides sequenced, reliable, bidirectional, connection-mode byte
43340 streams, and may provide a transmission mechanism for out-of-band
43341 data.

43342 SOCK_DGRAM Provides datagrams, which are connectionless-mode, unreliable messages
43343 of fixed maximum length.

43344 SOCK_SEQPACKET Provides sequenced, reliable, bidirectional, connection-mode
43345 transmission paths for records. A record can be sent using one or more
43346 output operations and received using one or more input operations, but a
43347 single operation never transfers part of more than one record. Record
43348 boundaries are visible to the receiver via the MSG_EOR flag.

43349 If the *protocol* argument is non-zero, it shall specify a protocol that is supported by the address
43350 family. If the *protocol* argument is zero, the default protocol for this address family and type shall
43351 be used. The protocols supported by the system are implementation-defined.

43352 The process may need to have appropriate privileges to use the *socketpair*( ) function or to create
43353 some sockets.

43354 **RETURN VALUE**

43355 Upon successful completion, this function shall return 0; otherwise, –1 shall be returned and
43356 *errno* set to indicate the error.

43357 **ERRORS**

43358 The *socketpair*( ) function shall fail if:

43359 [EAFNOSUPPORT]
43360 The implementation does not support the specified address family.

[EMFILE] No more file descriptors are available for this process.

[ENFILE] No more file descriptors are available for the system.

[EOPNOTSUPP] The specified protocol does not permit creation of socket pairs.

[EPROTONOSUPPORT]
The protocol is not supported by the address family, or the protocol is not supported by the implementation.

[EPROTOTYPE] The socket type is not supported by the protocol.

The *socketpair*( ) function may fail if:

[EACCES] The process does not have appropriate privileges.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

EXAMPLES

None.

APPLICATION USAGE

The documentation for specific address families specifies which protocols each address family supports. The documentation for specific protocols specifies which socket types each protocol supports.

The *socketpair*( ) function is used primarily with UNIX domain sockets and need not be supported for other domains.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

socket( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

43388 **NAME**

43389 sprintf — print formatted output

43390 **SYNOPSIS**

43391 `#include <stdio.h>`

43392 `int sprintf(char *restrict s, const char *restrict format, ...);`

43393 **DESCRIPTION**

43394 Refer to *fprintf*( ).

43395 **NAME**

43396 sqrt, sqrtf, sqrtl — square root function

43397 **SYNOPSIS**

43398
```
#include <math.h>
```

43399
```
double sqrt(double x);
```
43400
```
float sqrtf(float x);
```
43401
```
long double sqrtl(long double x);
```

43402 **DESCRIPTION**

43403 CX The functionality described on this reference page is aligned with the ISO C standard. Any
43404 conflict between the requirements described here and the ISO C standard is unintentional. This
43405 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

43406 These functions shall compute the square root of their argument x, $\sqrt{x}$.

43407 An application wishing to check for error situations should set *errno* to zero and call
43408 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
43409 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
43410 zero, an error has occurred.

43411 **RETURN VALUE**

43412 Upon successful completion, these functions shall return the square root of x.

43413 MX For finite values of $x < -0$, a domain error shall occur, and either a NaN (if supported), or an
43414 implementation-defined value shall be returned.

43415 MX If x is NaN, a NaN shall be returned.

43416 If x is ±0, or +Inf, x shall be returned.

43417 If x is –Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-
43418 defined value shall be returned.

43419 **ERRORS**

43420 These functions shall fail if:

43421 MX Domain Error The finite value of x is < -0, or x is –Inf.

43422 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
43423 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling |
43424 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception |
43425 shall be raised. |

43426 **EXAMPLES**

43427 **Taking the Square Root of 9.0**

43428
```
#include <math.h>
```
43429
```
...
```
43430
```
double x = 9.0;
```
43431
```
double result;
```
43432
```
...
```
43433
```
result = sqrt(x);
```

43434 **APPLICATION USAGE**

43435 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
43436 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

43437 **RATIONALE**

43438 None.

43439 **FUTURE DIRECTIONS**

43440 None.

43441 **SEE ALSO**

43442 *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, |
43443 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**, **<stdio.h>** |

43444 **CHANGE HISTORY**

43445 First released in Issue 1. Derived from Issue 1 of the SVID.

43446 **Issue 5**

43447 The DESCRIPTION is updated to indicate how an application should check for an error. This
43448 text was previously published in the APPLICATION USAGE section.

43449 **Issue 6**

43450 The *sqrtf*( ) and *sqrtl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

43451 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
43452 revised to align with the ISO / IEC 9899: 1999 standard.

43453 IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are
43454 marked.

43455 **NAME**

43456 srand — pseudo-random number generator

43457 **SYNOPSIS**

43458 `#include <stdlib.h>`

43459 `void srand(unsigned` *`seed`*`);`

43460 **DESCRIPTION**

43461 Refer to *rand*( ).

43462 **NAME**

43463 srand48 — seed uniformly distributed double-precision pseudo-random number generator

43464 **SYNOPSIS**

43465 XSI
```
#include <stdlib.h>

void srand48(long seedval);
```
43466

43467

43468 **DESCRIPTION**

43469 Refer to *drand48*( ).

43470 **NAME**

43471 srandom — seed pseudo-random number generator

43472 **SYNOPSIS**

43473 XSI
```
#include <stdlib.h>

void srandom(unsigned seed);
```
43474

43475

43476 **DESCRIPTION**

43477 Refer to *initstate*( ).

43478 **NAME**

43479 sscanf — convert formatted input

43480 **SYNOPSIS**

43481 `#include <stdio.h>`

43482 `int sscanf(const char *restrict s, const char *restrict format, ...);`

43483 **DESCRIPTION**

43484 Refer to *fscanf*( ).

43485 **NAME**

43486 stat — get file status

43487 **SYNOPSIS**

43488 `#include <sys/stat.h>`

43489 `int stat(const char *restrict path, struct stat *restrict buf);`

43490 **DESCRIPTION**

43491 The *stat*( ) function shall obtain information about the named file and write it to the area pointed
43492 to by the *buf* argument. The *path* argument points to a pathname naming a file. Read, write, or |
43493 execute permission of the named file is not required. An implementation that provides |
43494 additional or alternate file access control mechanisms may, under implementation-defined
43495 conditions, cause *stat*( ) to fail. In particular, the system may deny the existence of the file
43496 specified by *path*.

43497 If the named file is a symbolic link, the *stat*( ) function shall continue pathname resolution using |
43498 the contents of the symbolic link, and shall return information pertaining to the resulting file if |
43499 the file exists.

43500 The *buf* argument is a pointer to a **stat** structure, as defined in the **<sys/stat.h>** header, into
43501 which information is placed concerning the file.

43502 The *stat*( ) function shall update any time-related fields (as described in the Base Definitions
43503 volume of IEEE Std 1003.1-200x, Section 4.7, File Times Update), before writing into the **stat**
43504 structure.

43505 The structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime*
43506 shall have meaningful values for all file types defined in this volume of IEEE Std 1003.1-200x.
43507 The value of the member *st_nlink* shall be set to the number of links to the file.

43508 **RETURN VALUE**

43509 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
43510 indicate the error.

43511 **ERRORS**

43512 The *stat*( ) function shall fail if:

43513 [EACCES] Search permission is denied for a component of the path prefix.

43514 [EIO] An error occurred while reading from the file system.

43515 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
43516 argument.

43517 [ENAMETOOLONG]
43518 The length of the *path* argument exceeds {PATH_MAX} or a pathname |
43519 component is longer than {NAME_MAX}. |

43520 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

43521 [ENOTDIR] A component of the path prefix is not a directory.

43522 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file
43523 serial number cannot be represented correctly in the structure pointed to by
43524 *buf*.

43525 The *stat*( ) function may fail if:

43526 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
43527 resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

[EOVERFLOW] A value to be stored would overflow one of the members of the **stat** structure.

EXAMPLES

Obtaining File Status Information

The following example shows how to obtain file status information for a file named **/home/cnd/mod1**. The structure variable *buffer* is defined for the **stat** structure.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

struct stat buffer;
int         status;
...
status = stat("/home/cnd/mod1", &buffer);
```

Getting Directory Information

The following example fragment gets status information for each entry in a directory. The call to the *stat*( ) function stores file information in the **stat** structure pointed to by *statbuf*. The lines that follow the *stat*( ) call format the fields in the **stat** structure for presentation to the user of the program.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <locale.h>
#include <langinfo.h>
#include <stdio.h>
#include <stdint.h>

struct dirent  *dp;
struct stat     statbuf;
struct passwd  *pwd;
struct group   *grp;
struct tm      *tm;
char            datestring[256];
...
/* Loop through directory entries */
while ((dp = readdir(dir)) != NULL) {

    /* Get entry's information. */
    if (stat(dp->d_name, &statbuf) == -1)
        continue;

    /* Print out type, permissions, and number of links. */
    printf("%10.10s", sperm (statbuf.st_mode));
    printf("%4d", statbuf.st_nlink);
```

```
        /* Print out owners name if it is found using getpwuid(). */
        if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
            printf(" %-8.8s", pwd->pw_name);
        else
            printf(" %-8d", statbuf.st_uid);

        /* Print out group name if it's found using getgrgid(). */
        if ((grp = getgrgid(statbuf.st_gid)) != NULL)
            printf(" %-8.8s", grp->gr_name);
        else
            printf(" %-8d", statbuf.st_gid);

        /* Print size of file. */
        printf(" %9jd", (intmax_t)statbuf.st_size);

        tm = localtime(&statbuf.st_mtime);

        /* Get localized date string. */
        strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);

        printf(" %s %s\n", datestring, dp->d_name);
    }
```

APPLICATION USAGE

None.

RATIONALE

The intent of the paragraph describing ''additional or alternate file access control mechanisms'' is to allow a secure implementation where a process with a label that does not dominate the file's label cannot perform a *stat*( ) function. This is not related to read permission; a process with a label that dominates the file's label does not need read permission. An implementation that supports write-up operations could fail *fstat*( ) function calls even though it has a valid file descriptor open for writing.

FUTURE DIRECTIONS

None.

SEE ALSO

fstat( ), *lstat*( ), *readlink*( ), *symlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [EIO] mandatory error condition is added.

- The [ELOOP] mandatory error condition is added.
- The [EOVERFLOW] mandatory error condition is added. This change is to support large files.
- The [ENAMETOOLONG] and the second [EOVERFLOW] optional error conditions are added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Details are added regarding the treatment of symbolic links.
- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *stat*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

43627 **NAME**

43628 statvfs — get file system information

43629 **SYNOPSIS**

43630 XSI
```
#include <sys/statvfs.h>
```

43631
```
int statvfs(const char *restrict path, struct statvfs *restrict buf);
```

43632

43633 **DESCRIPTION**

43634 Refer to *fstatvfs*( ).

43635 **NAME**

43636 stderr, stdin, stdout — standard I/O streams

43637 **SYNOPSIS**

43638
```
#include <stdio.h>
```

43639
```
extern FILE *stderr, *stdin, *stdout;
```

43640 **DESCRIPTION**

43641 CX The functionality described on this reference page is aligned with the ISO C standard. Any
43642 conflict between the requirements described here and the ISO C standard is unintentional. This
43643 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

43644 A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type
43645 **FILE**. The *fopen*( ) function shall create certain descriptive data for a stream and return a pointer
43646 to designate the stream in all further transactions. Normally, there are three open streams with
43647 constant pointers declared in the **<stdio.h>** header and associated with the standard open files.

43648 At program start-up, three streams shall be predefined and need not be opened explicitly:
43649 *standard input* (for reading conventional input), *standard output* (for writing conventional output),
43650 and *standard error* (for writing diagnostic output). When opened, the standard error stream is not
43651 fully buffered; the standard input and standard output streams are fully buffered if and only if
43652 the stream can be determined not to refer to an interactive device.

43653 CX The following symbolic values in **<unistd.h>** define the file descriptors that shall be associated
43654 with the C-language *stdin*, *stdout*, and *stderr* when the application is started:

43655 STDIN_FILENO Standard input value, *stdin*. Its value is 0.

43656 STDOUT_FILENO Standard output value, *stdout*. Its value is 1.

43657 STDERR_FILENO Standard error value, *stderr*. Its value is 2.

43658 The *stderr* stream is expected to be open for reading and writing.

43659 **RETURN VALUE**

43660 None.

43661 **ERRORS**

43662 No errors are defined.

43663 **EXAMPLES**

43664 None.

43665 **APPLICATION USAGE**

43666 None.

43667 **RATIONALE**

43668 None.

43669 **FUTURE DIRECTIONS**

43670 None.

43671 **SEE ALSO**

43672 *fclose*( ), *feof*( ), *ferror*( ), *fileno*( ), *fopen*( ), *fread*( ), *fseek*( ), *getc*( ), *gets*( ), *popen*( ), *printf*( ), *putc*( ),
43673 *puts*( ), *read*( ), *scanf*( ), *setbuf*( ), *setvbuf*( ), *tmpfile*( ), *ungetc*( ), *vprintf*( ), the Base Definitions
43674 volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<unistd.h>**

43675 **CHANGE HISTORY**

43676 First released in Issue 1.

43677 **Issue 6**

43678 Extensions beyond the ISO C standard are now marked.

43679 A note that *stderr* is expected to be open for reading and writing is added to the DESCRIPTION.

43680 **NAME**

43681 strcasecmp, strncasecmp — case-insensitive string comparisons

43682 **SYNOPSIS**

43683 XSI
```
#include <strings.h>

int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t n);
```
43686

43687 **DESCRIPTION**

43688 The *strcasecmp*( ) function shall compare, while ignoring differences in case, the string pointed to
43689 by *s1* to the string pointed to by *s2*. The *strncasecmp*( ) function shall compare, while ignoring
43690 differences in case, not more than *n* bytes from the string pointed to by *s1* to the string pointed to
43691 by *s2*.

43692 In the POSIX locale, *strcasecmp*( ) and *strncasecmp*( ) shall behave as if the strings had been |
43693 converted to lowercase and then a byte comparison performed. The results are unspecified in |
43694 other locales. |

43695 **RETURN VALUE**

43696 Upon completion, *strcasecmp*( ) shall return an integer greater than, equal to, or less than 0, if the
43697 string pointed to by *s1* is, ignoring case, greater than, equal to, or less than the string pointed to
43698 by *s2*, respectively.

43699 Upon successful completion, *strncasecmp*( ) shall return an integer greater than, equal to, or less
43700 than 0, if the possibly null-terminated array pointed to by *s1* is, ignoring case, greater than, equal
43701 to, or less than the possibly null-terminated array pointed to by *s2*, respectively.

43702 **ERRORS**

43703 No errors are defined.

43704 **EXAMPLES**

43705 None.

43706 **APPLICATION USAGE**

43707 None.

43708 **RATIONALE**

43709 None.

43710 **FUTURE DIRECTIONS**

43711 None.

43712 **SEE ALSO**

43713 The Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

43714 **CHANGE HISTORY**

43715 First released in Issue 4, Version 2.

43716 **Issue 5**

43717 Moved from X/OPEN UNIX extension to BASE.

NAME

strcat — concatenate two strings

SYNOPSIS

```
#include <string.h>

char *strcat(char *restrict s1, const char *restrict s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strcat*( ) function shall append a copy of the string pointed to by *s2* (including the terminating null byte) to the end of the string pointed to by *s1*. The initial byte of *s2* overwrites the null byte at the end of *s1*. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *strcat*( ) function shall return *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

This issue is aligned with the ISO C standard; this does not affect compatibility with XPG3 applications. Reliable error detection by this function was never guaranteed.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strncat( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The *strcat*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

strchr — string scanning operation

SYNOPSIS

```
#include <string.h>

char *strchr(const char *s, int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *strchr*( ) function shall locate the first occurrence of *c* (converted to an **unsigned char**) in the string pointed to by *s*. The terminating null byte is considered to be part of the string.

RETURN VALUE

Upon completion, *strchr*( ) shall return a pointer to the byte, or a null pointer if the byte was not found.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strrchr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

strcmp — compare two strings

SYNOPSIS

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strcmp*( ) function shall compare the string pointed to by *s1* to the string pointed to by *s2*.

The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings being compared.

RETURN VALUE

Upon completion, *strcmp*( ) shall return an integer greater than, equal to, or less than 0, if the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*, respectively.

ERRORS

No errors are defined.

EXAMPLES

Checking a Password Entry

The following example compares the information read from standard input to the value of the name of the user entry. If the *strcmp*( ) function returns 0 (indicating a match), a further check will be made to see if the user entered the proper old password. The *crypt*( ) function shall encrypt the old password entered by the user, using the value of the encrypted password in the **passwd** structure as the salt. If this value matches the value of the encrypted **passwd** in the structure, the entered password *oldpasswd* is the correct user's password. Finally, the program encrypts the new password so that it can store the information in the **passwd** structure.

```
#include <string.h>
#include <unistd.h>
#include <stdio.h>
...
int valid_change;
struct passwd *p;
char user[100];
char oldpasswd[100];
char newpasswd[100];
char savepasswd[100];
...
if (strcmp(p->pw_name, user) == 0) {
    if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
        strcpy(savepasswd, crypt(newpasswd, user));
        p->pw_passwd = savepasswd;
        valid_change = 1;
    }
    else {
```

```
                fprintf(stderr, "Old password is not valid\n");
            }
        }
        ...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strncmp( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

strcoll — string comparison using collating information

SYNOPSIS

```
#include <string.h>

int strcoll(const char *s1, const char *s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strcoll*( ) function shall compare the string pointed to by *s1* to the string pointed to by *s2*, both interpreted as appropriate to the *LC_COLLATE* category of the current locale.

CX The *strcoll*( ) function shall not change the setting of *errno* if successful.

Since no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *strcoll*( ), then check *errno*.

RETURN VALUE

Upon successful completion, *strcoll*( ) shall return an integer greater than, equal to, or less than 0, according to whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2* when both are interpreted as appropriate to the current locale. On error, *strcoll*( ) may set *errno*, but no return value is reserved to indicate an error.

ERRORS

The *strcoll*( ) function may fail if:

CX [EINVAL] The *s1* or *s2* arguments contain characters outside the domain of the collating sequence.

EXAMPLES

Comparing Nodes

The following example uses an application-defined function, *node_compare*( ), to compare two nodes based on an alphabetical ordering of the *string* field.

```
#include <string.h>
...
struct node { /* These are stored in the table. */
    char *string;
    int length;
};
...
int node_compare(const void *node1, const void *node2)
{
    return strcoll(((const struct node *)node1)->string,
        ((const struct node *)node2)->string);
}
...
```

APPLICATION USAGE

The *strxfrm*( ) and *strcmp*( ) functions should be used for sorting large lists.

43884 **RATIONALE**

43885 None.

43886 **FUTURE DIRECTIONS**

43887 None.

43888 **SEE ALSO**

43889 *strcmp*( ), *strxfrm*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

43890 **CHANGE HISTORY**

43891 First released in Issue 3.

43892 **Issue 5**

43893 The DESCRIPTION is updated to indicate that *errno* does not be changed if the function is
43894 successful.

43895 **Issue 6**

43896 Extensions beyond the ISO C standard are now marked.

43897 The following new requirements on POSIX implementations derive from alignment with the
43898 Single UNIX Specification:

43899 • The [EINVAL] optional error condition is added.

43900 An example is added. |

NAME

strcpy — copy a string

SYNOPSIS

```
#include <string.h>

char *strcpy(char *restrict s1, const char *restrict s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strcpy*( ) function shall copy the string pointed to by *s2* (including the terminating null byte) into the array pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *strcpy*( ) function shall return *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

Initializing a String

The following example copies the string "----------" into the *permstring* variable.

```
#include <string.h>
...
static char permstring[11];
...
strcpy(permstring, "----------");
...
```

Storing a Key and Data

The following example allocates space for a key using *malloc*( ) then uses *strcpy*( ) to place the key there. Then it allocates space for data using *malloc*( ), and uses *strcpy*( ) to place data there. (The user-defined function *dbfree*( ) frees memory previously allocated to an array of type **struct element ***.)

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
...
/* Structure used to read data and store it. */
struct element {
    char *key;
    char *data;
};

struct element *tbl, *curtbl;
char *key, *data;
int count;
...
void dbfree(struct element *, int);
```

```
43945       ...
43946       if ((curtbl->key = malloc(strlen(key) + 1)) == NULL) {
43947           perror("malloc"); dbfree(tbl, count); return NULL;
43948       }
43949       strcpy(curtbl->key, key);
43950       if ((curtbl->data = malloc(strlen(data) + 1)) == NULL) {
43951           perror("malloc"); free(curtbl->key); dbfree(tbl, count); return NULL;
43952       }
43953       strcpy(curtbl->data, data);
43954       ...
```

APPLICATION USAGE

Character movement is performed differently in different implementations. Thus, overlapping moves may yield surprises.

This issue is aligned with the ISO C standard; this does not affect compatibility with XPG3 applications. Reliable error detection by this function was never guaranteed.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strncpy( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The *strcpy*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

strcspn — get length of a complementary substring

SYNOPSIS

```
#include <string.h>

size_t strcspn(const char *s1, const char *s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strcspn*( ) function shall compute the length (in bytes) of the maximum initial segment of the string pointed to by *s1* which consists entirely of bytes *not* from the string pointed to by *s2*.

RETURN VALUE

The *strcspn*( ) function shall return the length of the computed segment of the string pointed to by *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strspn( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The RETURN VALUE section is updated to indicated that *strcspn*( ) returns the length of *s1*, and not *s1* itself as was previously stated.

Issue 6

The Open Group Corrigendum U030/1 is applied. The text of the RETURN VALUE section is updated to indicate that the computed segment length is returned, not the *s1* length.

NAME

strdup — duplicate a string

SYNOPSIS

XSI
```
#include <string.h>

char *strdup(const char *s1);
```

DESCRIPTION

The *strdup*( ) function shall return a pointer to a new string, which is a duplicate of the string pointed to by *s1*. The returned pointer can be passed to *free*( ). A null pointer is returned if the new string cannot be created.

RETURN VALUE

The *strdup*( ) function shall return a pointer to a new string on success. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *strdup*( ) function may fail if:

[ENOMEM] Storage space available is insufficient.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

free( ), *malloc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

strerror, strerror_r — get error message string

SYNOPSIS

```
#include <string.h>

char *strerror(int errnum);
TSF int strerror_r(int errnum, char *strerrbuf, size_t buflen);
```

DESCRIPTION

CX For *strerror*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strerror*( ) function shall map the error number in *errnum* to a locale-dependent error message string and shall return a pointer to it. Typically, the values for *errnum* come from *errno*, but *strerror*( ) shall map any value of type **int** to a message.

The string pointed to shall not be modified by the application, but may be overwritten by a CX subsequent call to *strerror*( ) or *perror*( ).

CX The contents of the error message strings returned by *strerror*( ) should be determined by the setting of the *LC_MESSAGES* category in the current locale.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *strerror*( ).

CX The *strerror*( ) function shall not change the setting of *errno* if successful.

Since no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *strerror*( ), then check *errno*.

The *strerror*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *strerror_r*( ) function shall map the error number in *errnum* to a locale-dependent error message string and shall return the string in the buffer pointed to by *strerrbuf*, with length *buflen*.

RETURN VALUE

Upon successful completion, *strerror*( ) shall return a pointer to the generated message string. On error *errno* may be set, but no return value is reserved to indicate an error.

TSF Upon successful completion, *strerror_r*( ) shall return 0. Otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL] The value of *errnum* is not a valid error number.

The *strerror_r*( ) function may fail if:

TSF [ERANGE] Insufficient storage was supplied via *strerrbuf* and *buflen* to contain the generated message string.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

perror( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the fact that *errno* may be set is added.
- The [EINVAL] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *strerror_r*( ) function is added in response to IEEE PASC Interpretation 1003.1c #39.

The *strerror_r*( ) function is marked as part of the Thread-Safe Functions option.

NAME

strfmon — convert monetary value to a string

SYNOPSIS

XSI
```
#include <monetary.h>

ssize_t strfmon(char *restrict s, size_t maxsize,
    const char *restrict format, ...);
```

DESCRIPTION

The *strfmon*() function shall place characters into the array pointed to by *s* as controlled by the string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

The format is a character string, beginning and ending in its initial state, if any, that contains two types of objects: *plain characters*, which are simply copied to the output stream, and *conversion specifications*, each of which shall result in the fetching of zero or more arguments which are converted and formatted. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are simply ignored.

The application shall ensure that a conversion specification consists of the following sequence:

- A '%' character
- Optional flags
- Optional field width
- Optional left precision
- Optional right precision
- A required conversion specifier character that determines the conversion to be performed

Flags

One or more of the following optional flags can be specified to control the conversion:

=f An '=' followed by a single character *f* which is used as the numeric fill character. In order to work with precision or width counts, the fill character shall be a single byte character; if not, the behavior is undefined. The default numeric fill character is the <space>. This flag does not affect field width filling which always uses the <space>. This flag is ignored unless a left precision (see below) is specified.

^ Do not format the currency amount with grouping characters. The default is to insert the grouping characters if defined for the current locale.

+ or (Specify the style of representing positive and negative currency amounts. Only one of '+' or '(' may be specified. If '+' is specified, the locale's equivalent of '+' and '−' are used (for example, in the U.S., the empty string if positive and '−' if negative). If '(' is specified, negative amounts are enclosed within parentheses. If neither flag is specified, the '+' style is used.

! Suppress the currency symbol from the output conversion.

− Specify the alignment. If this flag is present the result of the conversion is left-justified (padded to the right) rather than right-justified. This flag shall be ignored unless a field width (see below) is specified.

Field Width

w A decimal digit string *w* specifying a minimum field width in bytes in which the result of the conversion is right-justified (or left-justified if the flag '−' is specified). The default is 0.

Left Precision

#n A '#' followed by a decimal digit string *n* specifying a maximum number of digits expected to be formatted to the left of the radix character. This option can be used to keep the formatted output from multiple calls to the *strfmon*( ) function aligned in the same columns. It can also be used to fill unused positions with a special character as in "$***123.45". This option causes an amount to be formatted as if it has the number of digits specified by *n*. If more than *n* digit positions are required, this conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character (see the *=f* flag above).

If grouping has not been suppressed with the '^' flag, and it is defined for the current locale, grouping separators are inserted before the fill characters (if any) are added. Grouping separators are not applied to fill characters even if the fill character is a digit.

To ensure alignment, any characters appearing before or after the number in the formatted output such as currency or sign symbols are padded as necessary with <space>s to make their positive and negative formats an equal length.

Right Precision

.p A period followed by a decimal digit string *p* specifying the number of digits after the radix character. If the value of the right precision *p* is 0, no radix character appears. If a right precision is not included, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

Conversion Specifier Characters

The conversion specifier characters and their meanings are:

i The **double** argument is formatted according to the locale's international currency format (for example, in the U.S.: USD 1,234.56). If the argument is ±Inf or NaN, the result of the conversion is unspecified.

n The **double** argument is formatted according to the locale's national currency format (for example, in the U.S.: $1,234.56). If the argument is ±Inf or NaN, the result of the conversion is unspecified.

% Convert to a '%'; no argument is converted. The entire conversion specification shall be %%.

Locale Information

The *LC_MONETARY* category of the program's locale affects the behavior of this function including the monetary radix character (which may be different from the numeric radix character affected by the *LC_NUMERIC* category), the grouping separator, the currency symbols, and formats. The international currency symbol should be conformant with the ISO 4217: 1995 standard.

If the value of *maxsize* is greater than {SSIZE_MAX}, the result is implementation-defined.

RETURN VALUE

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, *strfmon*( ) shall return the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, –1 shall be returned, the contents of the array are unspecified, and *errno* shall be set to indicate the error.

ERRORS

The *strfmon*( ) function shall fail if:

[E2BIG] Conversion stopped due to lack of space in the buffer.

EXAMPLES

Given a locale for the U.S. and the values 123.45, –123.45, and 3456.781:

Conversion Specification	Output	Comments
%n	$123.45 -$123.45 $3,456.78	Default formatting.
%11n	$123.45 -$123.45 $3,456.78	Right align within an 11 character field.
%#5n	$ 123.45 -$ 123.45 $ 3,456.78	Aligned columns for values up to 99,999.
%=*#5n	$***123.45 -$***123.45 $*3,456.78	Specify a fill character.
%=0#5n	$000123.45 -$000123.45 $03,456.78	Fill characters do not use grouping even if the fill character is a digit.
%^#5n	$ 123.45 -$ 123.45 $ 3456.78	Disable the grouping separator.
%^#5.0n	$ 123 -$ 123 $ 3457	Round off to whole units.
%^#5.4n	$ 123.4500 -$ 123.4500 $ 3456.7810	Increase the precision.
%(#5n	$ 123.45 ($ 123.45) $ 3,456.78	Use an alternative pos/neg style.
%(!#5n	123.45 (123.45) 3,456.78	Disable the currency symbol.
%-14#5.4n	$ 123.4500 -$ 123.4500 $3,456.7810	Left-justify the output.
%14#5.4n	$ 123.4500 -$ 123.4500 $3,456.7810	Corresponding right-justified output.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

Lowercase conversion characters are reserved for future standards use and uppercase for implementation-defined use.

SEE ALSO

localeconv( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<monetary.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 5

Moved from ENHANCED I18N to BASE.

The [ENOSYS] error is removed.

A sentence is added to the DESCRIPTION warning about values of *maxsize* that are greater than {SSIZE_MAX}.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *strfmon*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

strftime — convert date and time to a string

SYNOPSIS

```
#include <time.h>

size_t strftime(char *restrict s, size_t maxsize,
    const char *restrict format, const struct tm *restrict timeptr);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strftime*( ) function shall place bytes into the array pointed to by *s* as controlled by the string pointed to by *format*. The format is a character string, beginning and ending in its initial shift state, if any. The *format* string consists of zero or more conversion specifications and ordinary characters. A conversion specification consists of a '%' character, possibly followed by an E or O modifier, and a terminating conversion specifier character that determines the conversion specification's behavior. All ordinary characters (including the terminating null byte) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than *maxsize* bytes are placed into the array. Each conversion specifier is replaced by appropriate characters as described in the following list. The appropriate characters are determined using the *LC_TIME* category of the current locale and by the values of zero or more members of the broken-down time structure pointed to by *timeptr*, as specified in brackets in the description. If any of the specified values are outside the normal range, the characters stored are unspecified.

CX Local timezone information is used as though *strftime*( ) called *tzset*( ).

The following conversion specifications are supported:

%a Replaced by the locale's abbreviated weekday name. [*tm_wday*]

%A Replaced by the locale's full weekday name. [*tm_wday*]

%b Replaced by the locale's abbreviated month name. [*tm_mon*]

%B Replaced by the locale's full month name. [*tm_mon*]

%c Replaced by the locale's appropriate date and time representation. (See the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**.)

%C Replaced by the year divided by 100 and truncated to an integer, as a decimal number [00,99]. [*tm_year*]

%d Replaced by the day of the month as a decimal number [01,31]. [*tm_mday*]

%D Equivalent to %m/%d/%y. [*tm_mon*, *tm_mday*, *tm_year*]

%e Replaced by the day of the month as a decimal number [1,31]; a single digit is preceded by a space. [*tm_mday*]

%F Equivalent to %Y-%m-%d (the ISO 8601:2000 standard date format). [*tm_year*, *tm_mon*, *tm_mday*]

%g Replaced by the last 2 digits of the week-based year (see below) as a decimal number [00,99]. [*tm_year*, *tm_wday*, *tm_yday*]

%G Replaced by the week-based year (see below) as a decimal number (for example, 1977). [*tm_year*, *tm_wday*, *tm_yday*]

		%h	Equivalent to %b. [*tm_mon*]
		%H	Replaced by the hour (24-hour clock) as a decimal number [00,23]. [*tm_hour*]
		%I	Replaced by the hour (12-hour clock) as a decimal number [01,12]. [*tm_hour*]
		%j	Replaced by the day of the year as a decimal number [001,366]. [*tm_yday*]
		%m	Replaced by the month as a decimal number [01,12]. [*tm_mon*]
		%M	Replaced by the minute as a decimal number [00,59]. [*tm_min*]
		%n	Replaced by a <newline>.
		%p	Replaced by the locale's equivalent of either a.m. or p.m. [*tm_hour*]
	CX	%r	Replaced by the time in a.m. and p.m. notation; in the POSIX locale this shall be equivalent to %I:%M:%S %p. [*tm_hour*, *tm_min*, *tm_sec*]
		%R	Replaced by the time in 24 hour notation (%H:%M). [*tm_hour*, *tm_min*]
		%S	Replaced by the second as a decimal number [00,60]. [*tm_sec*]
		%t	Replaced by a <tab>.
		%T	Replaced by the time (%H:%M:%S). [*tm_hour*, *tm_min*, *tm_sec*]
		%u	Replaced by the weekday as a decimal number [1,7], with 1 representing Monday. [*tm_wday*]
		%U	Replaced by the week number of the year as a decimal number [00,53]. The frist Sunday of January is the first day of week 1; days in the new year before this are in week 0. [*tm_year*, *tm_wday*, *tm_yday*]
		%V	Replaced by the week number of the year (Monday as the first day of the week) as a decimal number [01,53]. If the week containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1. Both January 4th and the first Thursday of January are always in week 1. [*tm_year*, *tm_wday*, *tm_yday*]
		%w	Replaced by the weekday as a decimal number [0,6], with 0 representing Sunday. [*tm_wday*]
		%W	Replaced by the week number of the year as a decimal number [00,53]. The first Monday of January is the first day of week 1; days in the new year before this are in week 0. [*tm_year*, *tm_wday*, *tm_yday*]
		%x	Replaced by the locale's appropriate date representation. (See the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**.)
		%X	Replaced by the locale's appropriate time representation. (See the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**.)
		%y	Replaced by the last two digits of the year as a decimal number [00,99]. [*tm_year*]
		%Y	Replaced by the year as a decimal number (for example, 1997). [*tm_year*]
		%z	Replaced by the offset from UTC in the ISO 8601:2000 standard format (+hhmm or –hhmm), or by no characters if no timezone is determinable. For example, "–0430"
	CX		means 4 hours 30 minutes behind UTC (west of Greenwich). If *tm_isdst* is zero, the standard time offset is used. If *tm_isdst* is greater than zero, the daylight savings time offset is used. If *tm_isdst* is negative, no characters are returned. [*tm_isdst*]

%Z Replaced by the timezone name or abbreviation, or by no bytes if no timezone information exists. [*tm_isdst*]

%% Replaced by %.

If a conversion specification does not correspond to any of the above, the behavior is undefined.

Modified Conversion Specifiers

Some conversion specifiers can be modified by the E or O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier. If the alternative format or specification does not exist for the current locale, (see ERA in the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.5, LC_TIME) the behavior shall be as if the unmodified conversion specification were used.

%Ec Replaced by the locale's alternative appropriate date and time representation.

%EC Replaced by the name of the base year (period) in the locale's alternative representation.

%Ex Replaced by the locale's alternative date representation.

%EX Replaced by the locale's alternative time representation.

%Ey Replaced by the offset from %EC (year only) in the locale's alternative representation.

%EY Replaced by the full alternative year representation.

%Od Replaced by the day of the month, using the locale's alternative numeric symbols, filled as needed with leading zeros if there is any alternative symbol for zero; otherwise, with leading spaces.

%Oe Replaced by the day of the month, using the locale's alternative numeric symbols, filled as needed with leading spaces.

%OH Replaced by the hour (24-hour clock) using the locale's alternative numeric symbols.

%OI Replaced by the hour (12-hour clock) using the locale's alternative numeric symbols.

%Om Replaced by the month using the locale's alternative numeric symbols.

%OM Replaced by the minutes using the locale's alternative numeric symbols.

%OS Replaced by the seconds using the locale's alternative numeric symbols.

%Ou Replaced by the weekday as a number in the locale's alternative representation (Monday=1).

%OU Replaced by the week number of the year (Sunday as the first day of the week, rules corresponding to %U) using the locale's alternative numeric symbols.

%OV Replaced by the week number of the year (Monday as the first day of the week, rules corresponding to %V) using the locale's alternative numeric symbols.

%Ow Replaced by the number of the weekday (Sunday=0) using the locale's alternative numeric symbols.

%OW Replaced by the week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.

%Oy Replaced by the year (offset from %C) using the locale's alternative numeric symbols.

%g, %G, and %V give values according to the ISO 8601:2000 standard week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th,

which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, %G is replaced by 1998 and %V is replaced by 53. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, %G is replaced by 1998 and %V is replaced by 01.

If a conversion specifier is not one of the above, the behavior is undefined.

RETURN VALUE

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, *strftime*( ) shall return the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, 0 shall be returned and the contents of the array are unspecified.

ERRORS

No errors are defined.

EXAMPLES

Getting a Localized Date String

The following example first sets the locale to the user's default. The locale information will be used in the *nl_langinfo*( ) and *strftime*( ) functions. The *nl_langinfo*( ) function returns the localized date string which specifies how the date is laid out. The *strftime*( ) function takes this information and, using the **tm** structure for values, places the date and time information into *datestring*.

```
#include <time.h>
#include <locale.h>
#include <langinfo.h>
...
struct tm *tm;
char datestring[256];
...
setlocale (LC_ALL, "");
...
strftime (datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);
...
```

APPLICATION USAGE

The range of values for %S is [00,60] rather than [00,59] to allow for the occasional leap second.

Some of the conversion specifications are duplicates of others. They are included for compatibility with *nl_cxtime*( ) and *nl_ascxtime*( ), which were published in Issue 2.

Applications should use %Y (4-digit years) in preference to %y (2-digit years).

In the C locale, the E and O modifiers are ignored and the replacement strings for the following specifiers are:

%a The first three characters of %A.

%A One of Sunday, Monday, ..., Saturday.

%b The first three characters of %B.

%B One of January, February, ..., December.

%c Equivalent to %a %b %e %T %Y.

%p One of AM or PM.

%r Equivalent to %I:%M:%S %p.

%x Equivalent to %m/%d/%y.

%X Equivalent to %T.

%Z Implementation-defined.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime( ), *clock*( ), *ctime*( ), *difftime*( ), *getdate*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strptime*( ), *time*( ), *tzset*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

The description of %OV is changed to be consistent with %V and defines Monday as the first day of the week.

The description of %Oy is clarified.

Issue 6

Extensions beyond the ISO C standard are now marked.

The Open Group Corrigendum U033/8 is applied. The %V conversion specifier is changed from ''Otherwise, it is week 53 of the previous year, and the next week is week 1'' to ''Otherwise, it is the last week of the previous year, and the next week is week 1''.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The %C, %D, %e, %h, %n, %r, %R, %t, and %T conversion specifiers are added.
- The modified conversion specifiers are added for consistency with the ISO POSIX-2 standard *date* utility.

The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- The *strftime*( ) prototype is updated.
- The DESCRIPTION is extensively revised.
- The %z conversion specifier is added.

A new example is added.

NAME

strlen — get string length

SYNOPSIS

```
#include <string.h>

size_t strlen(const char *s);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strlen*() function shall compute the number of bytes in the string to which *s* points, not including the terminating null byte.

RETURN VALUE

The *strlen*() function shall return the length of *s*; no return value shall be reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

Getting String Lengths

The following example sets the maximum length of *key* and *data* by using *strlen*() to get the lengths of those strings.

```
#include <string.h>
...
struct element {
    char *key;
    char *data;
};
...
char *key, *data;
int len;

*keylength = *datalength = 0;
...
if ((len = strlen(key)) > *keylength)
    *keylength = len;
if ((len = strlen(data)) > *datalength)
    *datalength = len;
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The RETURN VALUE section is updated to indicate that *strlen*( ) returns the length of *s*, and not *s* itself as was previously stated.

NAME

strncasecmp — case-insensitive string comparison

SYNOPSIS

XSI

```
#include <strings.h>

int strncasecmp(const char *s1, const char *s2, size_t n);
```

DESCRIPTION

Refer to *strcasecmp*( ).

NAME

strncat — concatenate a string with part of another

SYNOPSIS

```
#include <string.h>

char *strncat(char *restrict s1, const char *restrict s2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strncat*( ) function shall append not more than *n* bytes (a null byte and bytes that follow it are not appended) from the array pointed to by *s2* to the end of the string pointed to by *s1*. The initial byte of *s2* overwrites the null byte at the end of *s1*. A terminating null byte is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *strncat*( ) function shall return *s1*; no return value shall be reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcat( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The *strncat*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

NAME

strncmp — compare part of two strings

SYNOPSIS

```
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strncmp*( ) function shall compare not more than *n* bytes (bytes that follow a null byte are not compared) from the array pointed to by *s1* to the array pointed to by *s2*.

The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings being compared.

RETURN VALUE

Upon successful completion, *strncmp*( ) shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by *s1* is greater than, equal to, or less than the possibly null-terminated array pointed to by *s2* respectively.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcmp( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

strncpy — copy part of a string

SYNOPSIS

```
#include <string.h>

char *strncpy(char *restrict s1, const char *restrict s2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strncpy*( ) function shall copy not more than *n* bytes (bytes that follow a null byte are not copied) from the array pointed to by *s2* to the array pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

If the array pointed to by *s2* is a string that is shorter than *n* bytes, null bytes shall be appended to the copy in the array pointed to by *s1*, until *n* bytes in all are written.

RETURN VALUE

The *strncpy*( ) function shall return *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Character movement is performed differently in different implementations. Thus, overlapping moves may yield surprises.

If there is no null byte in the first *n* bytes of the array pointed to by *s2*, the result is not null-terminated.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcpy( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The *strncpy*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

44605 **NAME**

44606 strpbrk — scan string for byte

44607 **SYNOPSIS**

44608
```
#include <string.h>
```

44609
```
char *strpbrk(const char *s1, const char *s2);
```

44610 **DESCRIPTION**

44611 CX The functionality described on this reference page is aligned with the ISO C standard. Any
44612 conflict between the requirements described here and the ISO C standard is unintentional. This
44613 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

44614 The *strpbrk*( ) function shall locate the first occurrence in the string pointed to by *s1* of any byte
44615 from the string pointed to by *s2*.

44616 **RETURN VALUE**

44617 Upon successful completion, *strpbrk*( ) shall return a pointer to the byte or a null pointer if no
44618 byte from *s2* occurs in *s1*.

44619 **ERRORS**

44620 No errors are defined.

44621 **EXAMPLES**

44622 None.

44623 **APPLICATION USAGE**

44624 None.

44625 **RATIONALE**

44626 None.

44627 **FUTURE DIRECTIONS**

44628 None.

44629 **SEE ALSO**

44630 *strchr*( ), *strrchr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

44631 **CHANGE HISTORY**

44632 First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

strptime — date and time conversion

SYNOPSIS

XSI
```
#include <time.h>

char *strptime(const char *restrict buf, const char *restrict format,
    struct tm *restrict tm);
```

DESCRIPTION

The *strptime*( ) function shall convert the character string pointed to by *buf* to values which are stored in the **tm** structure pointed to by *tm*, using the format specified by *format*.

The *format* is composed of zero or more directives. Each directive is composed of one of the following: one or more white-space characters (as specified by *isspace*( )); an ordinary character (neither '%' nor a white-space character); or a conversion specification. Each conversion specification is composed of a '%' character followed by a conversion character which specifies the replacement required. The application shall ensure that there is white-space or other non-alphanumeric characters between any two conversion specifications. The following conversion specifications are supported:

%a — The day of the week, using the locale's weekday names; either the abbreviated or full name may be specified.

%A — Equivalent to %a. |

%b — The month, using the locale's month names; either the abbreviated or full name may be specified.

%B — Equivalent to %b. |

%c — Replaced by the locale's appropriate date and time representation.

%C — The century number [0,99]; leading zeros are permitted but not required.

%d — The day of the month [1,31]; leading zeros are permitted but not required.

%D — The date as %m/%d/%y.

%e — Equivalent to %d. |

%h — Equivalent to %b. |

%H — The hour (24-hour clock) [0,23]; leading zeros are permitted but not required.

%I — The hour (12-hour clock) [1,12]; leading zeros are permitted but not required.

%j — The day number of the year [1,366]; leading zeros are permitted but not required.

%m — The month number [1,12]; leading zeros are permitted but not required.

%M — The minute [0,59]; leading zeros are permitted but not required. |

%n — Any white space.

%p — The locale's equivalent of a.m or p.m.

%r — 12-hour clock time using the AM/PM notation if **t_fmt_ampm** is not an empty string in the LC_TIME portion of the current locale; in the POSIX locale, this shall be equivalent to %I:%M:%S %p. | |

%R — The time as %H:%M.

%S	The seconds [0,60]; leading zeros are permitted but not required.
%t	Any white space.
%T	The time as %H:%M:%S.
%U	The week number of the year (Sunday as the first day of the week) as a decimal number [00,53]; leading zeros are permitted but not required.
%w	The weekday as a decimal number [0,6], with 0 representing Sunday; leading zeros are permitted but not required.
%W	The week number of the year (Monday as the first day of the week) as a decimal number [00,53]; leading zeros are permitted but not required.
%x	The date, using the locale's date format.
%X	The time, using the locale's time format.
%y	The year within century. When a century is not otherwise specified, values in the range [69,99] shall refer to years 1969 to 1999 inclusive, and values in the range [00,68] shall refer to years 2000 to 2068 inclusive; leading zeros shall be permitted but shall not be required. **Note:** It is expected that in a future version of IEEE Std 1003.1-200x the default century inferred from a 2-digit year will change. (This would apply to all commands accepting a 2-digit year as input.)
%Y	The year, including the century (for example, 1988).
%%	Replaced by %.

Modified Conversion Specifiers

Some conversion specifiers can be modified by the E and O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier. If the alternative format or specification does not exist in the current locale, the behavior shall be as if the unmodified conversion specification were used.

%Ec	The locale's alternative appropriate date and time representation.
%EC	The name of the base year (period) in the locale's alternative representation.
%Ex	The locale's alternative date representation.
%EX	The locale's alternative time representation.
%Ey	The offset from %EC (year only) in the locale's alternative representation.
%EY	The full alternative year representation.
%Od	The day of the month using the locale's alternative numeric symbols; leading zeros are permitted but not required.
%Oe	Equivalent to %Od.
%OH	The hour (24-hour clock) using the locale's alternative numeric symbols.
%OI	The hour (12-hour clock) using the locale's alternative numeric symbols.
%Om	The month using the locale's alternative numeric symbols.
%OM	The minutes using the locale's alternative numeric symbols.

%OS The seconds using the locale's alternative numeric symbols.

%OU The week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.

%Ow The number of the weekday (Sunday=0) using the locale's alternative numeric symbols.

%OW The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.

%Oy The year (offset from %C) using the locale's alternative numeric symbols.

A conversion specification composed of white-space characters is executed by scanning input up to the first character that is not white-space (which remains unscanned), or until no more characters can be scanned.

A conversion specification that is an ordinary character is executed by scanning the next character from the buffer. If the character scanned from the buffer differs from the one comprising the directive, the directive fails, and the differing and subsequent characters remain unscanned.

A series of conversion specifications composed of %n, %t, white-space characters, or any combination is executed by scanning up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.

Any other conversion specification is executed by scanning characters until a character matching the next directive is scanned, or until no more characters can be scanned. These characters, except the one matching the next directive, are then compared to the locale values associated with the conversion specifier. If a match is found, values for the appropriate **tm** structure members are set to values corresponding to the locale information. Case is ignored when matching items in *buf* such as month or weekday names. If no match is found, *strptime*() fails and no more characters are scanned.

RETURN VALUE

Upon successful completion, *strptime*() shall return a pointer to the character following the last character parsed. Otherwise, a null pointer shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Several ''equivalent to'' formats and the special processing of white-space characters are provided in order to ease the use of identical *format* strings for *strftime*() and *strptime*().

Applications should use %Y (4-digit years) in preference to %y (2-digit years).

It is unspecified whether multiple calls to *strptime*() using the same **tm** structure will update the current contents of the structure or overwrite all contents of the structure. Conforming applications should make a single call to *strptime*() with a format and all data needed to completely specify the date and time being converted.

RATIONALE

None.

FUTURE DIRECTIONS

The *strptime*( ) function is expected to be mandatory in the next version of this volume of IEEE Std 1003.1-200x.

SEE ALSO

scanf( ), *strftime*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 5

Moved from ENHANCED I18N to BASE.

The [ENOSYS] error is removed.

The exact meaning of the %y and %Oy specifiers are clarified in the DESCRIPTION.

Issue 6

The Open Group Corrigendum U033/5 is applied. The %r specifier description is reworded.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *strptime*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

The Open Group Corrigendum U047/2 is applied.

The DESCRIPTION is updated to use the terms ‘‘conversion specifier’’ and ‘‘conversion specification’’ for consistency with *strftime*( ).

NAME

strrchr — string scanning operation

SYNOPSIS

```
#include <string.h>

char *strrchr(const char *s, int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *strrchr*( ) function shall locate the last occurrence of *c* (converted to an **unsigned char**) in the string pointed to by *s*. The terminating null byte is considered to be part of the string.

RETURN VALUE

Upon successful completion, *strrchr*( ) shall return a pointer to the byte or a null pointer if *c* does not occur in the string.

ERRORS

No errors are defined.

EXAMPLES

Finding the Base Name of a File

The following example uses *strrchr*( ) to get a pointer to the base name of a file. The *strrchr*( ) function searches backwards through the name of the file to find the last `'/'` character in *name*. This pointer (plus one) will point to the base name of the file.

```
#include <string.h>
...
const char *name;
char *basename;
...
basename = strrchr(name, '/') + 1;
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strchr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

strspn — get length of a substring

SYNOPSIS

```
#include <string.h>

size_t strspn(const char *s1, const char *s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strspn*( ) function shall compute the length (in bytes) of the maximum initial segment of the string pointed to by *s1* which consists entirely of bytes from the string pointed to by *s2*.

RETURN VALUE

The *strspn*( ) function shall return the length of *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcspn( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The RETURN VALUE section is updated to indicate that *strspn*( ) returns the length of *s*, and not *s* itself as was previously stated.

NAME

strstr — find a substring

SYNOPSIS

```
#include <string.h>

char *strstr(const char *s1, const char *s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strstr*( ) function shall locate the first occurrence in the string pointed to by *s1* of the sequence of bytes (excluding the terminating null byte) in the string pointed to by *s2*.

RETURN VALUE

Upon successful completion, *strstr*( ) shall return a pointer to the located string or a null pointer if the string is not found.

If *s2* points to a string with zero length, the function shall return *s1*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strchr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the ANSI C standard.

NAME

strtod, strtof, strtold — convert string to a double-precision number

SYNOPSIS

```
#include <stdlib.h>

double strtod(const char *restrict nptr, char **restrict endptr);
float strtof(const char *restrict nptr, char **restrict endptr);
long double strtold(const char *restrict nptr, char **restrict endptr);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall convert the initial portion of the string pointed to by *nptr* to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace*( ))
2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string

Then they shall attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent part
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character, then an optional binary exponent part
- One of INF or INFINITY, ignoring case
- One of NAN or NAN(*n-char-sequence*$_{opt}$), ignoring case in the NAN part, where:

```
n-char-sequence:
    digit
    nondigit
    n-char-sequence digit
    n-char-sequence nondigit
```

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) shall be interpreted as a floating constant of the C language, except that the radix character shall be used in place of a period, and that if neither an exponent part nor a radix character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence shall be interpreted as negated. A character sequence INF or INFINITY shall be interpreted as an

infinity, if representable in the return type, else as if it were a floating constant that is too large for the range of the return type. A character sequence NAN or NAN(*n-char-sequence*$_{opt}$) shall be interpreted as a quiet NaN, if supported in the return type, else as if it were a subject sequence part that does not have the expected form; the meaning of the *n*-char sequences is implementation-defined. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the value resulting from the conversion is correctly rounded.

CX The radix character is defined in the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period ('.').

CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX The *strtod*( ) function shall not change the setting of *errno* if successful.

Since 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set *errno* to 0, then call *strtod*( ), *strtof*( ), or *strtold*( ), then check *errno*.

RETURN VALUE

Upon successful completion, these functions shall return the converted value. If no conversion could be performed, 0 shall be returned, and *errno* may be set to [EINVAL].

If the correct value is outside the range of representable values, HUGE_VAL, HUGE_VALF, or HUGE_VALL shall be returned (according to the sign of the value), and *errno* shall be set to [ERANGE].

If the correct value would cause an underflow, a value whose magnitude is no greater than the smallest normalized positive number in the return type shall be returned and *errno* set to [ERANGE].

ERRORS

These functions shall fail if:

CX [ERANGE] The value to be returned would cause overflow or underflow.

These functions may fail if:

CX [EINVAL] No conversion could be performed.

EXAMPLES

None.

APPLICATION USAGE

If the subject sequence has the hexadecimal form and FLT_RADIX is not a power of 2, adn the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

If the subject sequence has the decimal form and at most DECIMAL_DIG (defined in **<float.h>**) significant digits, the result should be correctly rounded. If the subject sequence *D* has the decimal form and more than DECIMAL_DIG significant digits, consider the two bounding, adjacent decimal strings *L* and *U*, both having DECIMAL_DIG significant digits, such that the

values of *L*, *D*, and *U* satisfy *L* <= *D* <= *U*. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding *L* and *U* according to the current rounding direction, with the extra stipulation that the error with respect to *D* should have a correct sign for the current rounding direction.

The changes to *strtod*( ) introduced by the ISO / IEC 9899: 1999 standard can alter the behavior of well-formed applications complying with the ISO / IEC 9899: 1990 standard and thus earlier versions of IEEE Std 1003.1-200x. One such example would be:

```
int
what_kind_of_number (char *s)
{
    char *endp;
    double d;
    long l;

    d = strtod(s, &endp);
    if (s != endp && *endp == '\0')
        printf("It's a float with value %g\n", d);
    else
    {
        l = strtol(s, &endp, 0);
        if (s != endp && *endp == '\0')
            printf("It's an integer with value %ld\n", 1);
        else
            return 1;
    }
    return 0;
}
```

If the function is called with:

```
what_kind_of_number ("0x10")
```

an ISO / IEC 9899: 1990 standard-compliant library will result in the function printing:

```
It's an integer with value 16
```

With the ISO / IEC 9899: 1999 standard, the result is:

```
It's a float with value 16
```

The change in behavior is due to the inclusion of floating-point numbers in hexadecimal notation without requiring that either a decimal point or the binary exponent be present.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isspace( ), *localeconv*( ), *scanf*( ), *setlocale*( ), *strtol*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<float.h>**, **<stdlib.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

45000 **CHANGE HISTORY**

45001 First released in Issue 1. Derived from Issue 1 of the SVID.

45002 **Issue 5**

45003 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

45004 **Issue 6**

45005 Extensions beyond the ISO C standard are now marked.

45006 The following new requirements on POSIX implementations derive from alignment with the
45007 Single UNIX Specification:

45008 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
45009 added if no conversion could be performed.

45010 The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

45011 • The *strtod*( ) function is updated.

45012 • The *strtof*( ) and *strtold*( ) functions are added.

45013 • The DESCRIPTION is extensively revised.

45014 ISO / IEC 9899: 1999 standard, Technical Corrigendum No. 1 is incorporated. |

45015 **NAME**

45016 strtof — convert string to a double-precision number

45017 **SYNOPSIS**

45018 `#include <stdlib.h>`

45019 `float strtof(const char *restrict nptr, char **restrict endptr);`

45020 **DESCRIPTION**

45021 Refer to *strtod*( ).

45022 **NAME**

45023 strtoimax, strtoumax — convert string to integer type

45024 **SYNOPSIS**

45025
```
#include <inttypes.h>
```

45026
```
intmax_t strtoimax(const char *restrict nptr, char **restrict endptr,
    int base);
```
45027
45028
```
uintmax_t strtoumax(const char *restrict nptr, char **restrict endptr,
    int base);
```
45029

45030 **DESCRIPTION**

45031 CX The functionality described on this reference page is aligned with the ISO C standard. Any
45032 conflict between the requirements described here and the ISO C standard is unintentional. This
45033 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45034 These functions shall be equivalent to the *strtol*( ), *strtoll*( ), *strtoul*( ), and *strtoull*( ) functions,
45035 except that the initial portion of the string shall be converted to **intmax_t** and **uintmax_t**
45036 representation, respectively.

45037 **RETURN VALUE**

45038 These functions shall return the converted value, if any.

45039 If no conversion could be performed, zero shall be returned.

45040 If the correct value is outside the range of representable values, {INTMAX_MAX},
45041 {INTMAX_MIN}, or {UINTMAX_MAX} shall be returned (according to the return type and sign
45042 of the value, if any), and *errno* shall be set to [ERANGE].

45043 **ERRORS**

45044 These functions shall fail if:

45045 [ERANGE] The value to be returned is not representable.

45046 These functions may fail if:

45047 [EINVAL] The value of *base* is not supported.

45048 **EXAMPLES**

45049 None.

45050 **APPLICATION USAGE**

45051 None.

45052 **RATIONALE**

45053 None.

45054 **FUTURE DIRECTIONS**

45055 None.

45056 **SEE ALSO**

45057 *strtol*( ), *strtoul*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<inttypes.h>**

45058 **CHANGE HISTORY**

45059 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

45060 **NAME**

45061 strtok, strtok_r — split string into tokens

45062 **SYNOPSIS**

45063
```
#include <string.h>
```

45064
```
char *strtok(char *restrict s1, const char *restrict s2);
```

45065 TSF
```
char *strtok_r(char *restrict s, const char *restrict sep,
    char **restrict lasts);
```

45066

45067

45068 **DESCRIPTION**

45069 CX For *strtok*(): The functionality described on this reference page is aligned with the ISO C
45070 standard. Any conflict between the requirements described here and the ISO C standard is
45071 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45072 A sequence of calls to *strtok*() breaks the string pointed to by *s1* into a sequence of tokens, each
45073 of which is delimited by a byte from the string pointed to by *s2*. The first call in the sequence has
45074 *s1* as its first argument, and is followed by calls with a null pointer as their first argument. The
45075 separator string pointed to by *s2* may be different from call to call.

45076 The first call in the sequence searches the string pointed to by *s1* for the first byte that is *not*
45077 contained in the current separator string pointed to by *s2*. If no such byte is found, then there
45078 are no tokens in the string pointed to by *s1* and *strtok*() shall return a null pointer. If such a byte
45079 is found, it is the start of the first token.

45080 The *strtok*() function then searches from there for a byte that *is* contained in the current
45081 separator string. If no such byte is found, the current token extends to the end of the string
45082 pointed to by *s1*, and subsequent searches for a token shall return a null pointer. If such a byte is
45083 found, it is overwritten by a null byte, which terminates the current token. The *strtok*() function
45084 saves a pointer to the following byte, from which the next search for a token shall start.

45085 Each subsequent call, with a null pointer as the value of the first argument, starts searching from
45086 the saved pointer and behaves as described above.

45087 The implementation shall behave as if no function defined in this volume of
45088 IEEE Std 1003.1-200x calls *strtok*().

45089 CX The *strtok*() function need not be reentrant. A function that is not required to be reentrant is not
45090 required to be thread-safe.

45091 TSF The *strtok_r*() function considers the null-terminated string *s* as a sequence of zero or more text
45092 tokens separated by spans of one or more characters from the separator string *sep*. The
45093 argument *lasts* points to a user-provided pointer which points to stored information necessary
45094 for *strtok_r*() to continue scanning the same string.

45095 In the first call to *strtok_r*(), *s* points to a null-terminated string, *sep* to a null-terminated string of
45096 separator characters, and the value pointed to by *lasts* is ignored. The *strtok_r*() function shall
45097 return a pointer to the first character of the first token, write a null character into *s* immediately
45098 following the returned token, and update the pointer to which *lasts* points.

45099 In subsequent calls, *s* is a NULL pointer and *lasts* shall be unchanged from the previous call so
45100 that subsequent calls shall move through the string *s*, returning successive tokens until no
45101 tokens remain. The separator string *sep* may be different from call to call. When no token
45102 remains in *s*, a NULL pointer shall be returned.

45103 **RETURN VALUE**

45104 Upon successful completion, *strtok*( ) shall return a pointer to the first byte of a token. Otherwise,
45105 if there is no token, *strtok*( ) shall return a null pointer.

45106 TSF The *strtok_r*( ) function shall return a pointer to the token found, or a NULL pointer when no
45107 token is found.

45108 **ERRORS**

45109 No errors are defined.

45110 **EXAMPLES**

45111 **Searching for Word Separators**

45112 The following example searches for tokens separated by space characters.

```
45113 #include <string.h>
45114 ...
45115 char *token;
45116 char *line = "LINE TO BE SEPARATED";
45117 char *search = " ";
45118 /* Token will point to "LINE". */
45119 token = strtok(line, search);
45120 /* Token will point to "TO". */
45121 token = strtok(NULL, search);
```

45122 **Breaking a Line**

45123 The following example uses *strtok*( ) to break a line into two character strings separated by any
45124 combination of <space>s, <tab>s, or <newline>s.

```
45125 #include <string.h>
45126 ...
45127 struct element {
45128     char *key;
45129     char *data;
45130 };
45131 ...
45132 char line[LINE_MAX];
45133 char *key, *data;
45134 ...
45135 key = strtok(line, " \n");
45136 data = strtok(NULL, " \n");
45137 ...
```

45138 **APPLICATION USAGE**

45139 The *strtok_r*( ) function is thread-safe and stores its state in a user-supplied buffer instead of
45140 possibly using a static data area that may be overwritten by an unrelated call from another
45141 thread.

45142 **RATIONALE**

45143 The *strtok*( ) function searches for a separator string within a larger string. It returns a pointer to
45144 the last substring between separator strings. This function uses static storage to keep track of
45145 the current string position between calls. The new function, *strtok_r*( ), takes an additional
45146 argument, *lasts*, to keep track of the current position in the string.

45147 **FUTURE DIRECTIONS**

45148 None.

45149 **SEE ALSO**

45150 The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

45151 **CHANGE HISTORY**

45152 First released in Issue 1. Derived from Issue 1 of the SVID.

45153 **Issue 5**

45154 The *strtok_r*( ) function is included for alignment with the POSIX Threads Extension.

45155 A note indicating that the *strtok*( ) function need not be reentrant is added to the DESCRIPTION.

45156 **Issue 6**

45157 Extensions beyond the ISO C standard are now marked.

45158 The *strtok_r*( ) function is marked as part of the Thread-Safe Functions option.

45159 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

45160 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
45161 its avoidance of possibly using a static data area.

45162 The **restrict** keyword is added to the *strtok*( ) and *strtok_r*( ) prototypes for alignment with the
45163 ISO / IEC 9899: 1999 standard.

45164 **NAME**

45165 strtol, strtoll — convert string to a long integer

45166 **SYNOPSIS**

45167 ```
#include <stdlib.h>
```

45168 ```
long strtol(const char *restrict str, char **restrict endptr, int base);
```
45169 ```
long long strtoll(const char *restrict str, char **restrict endptr,
```
45170 ```
    int base)
```

45171 **DESCRIPTION**

45172 CX The functionality described on this reference page is aligned with the ISO C standard. Any
45173 conflict between the requirements described here and the ISO C standard is unintentional. This
45174 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45175 These functions shall convert the initial portion of the string pointed to by *str* to a type **long** and
45176 **long long** representation, respectively. First, they decompose the input string into three parts:

45177 1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace*( ))

45178 2. A subject sequence interpreted as an integer represented in some radix determined by the
45179 value of *base*

45180 3. A final string of one or more unrecognized characters, including the terminating null byte
45181 of the input string.

45182 Then they shall attempt to convert the subject sequence to an integer, and return the result. |

45183 If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant,
45184 octal constant, or hexadecimal constant, any of which may be preceded by a `'+'` or `'−'` sign. A
45185 decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An
45186 octal constant consists of the prefix `'0'` optionally followed by a sequence of the digits `'0'` to
45187 `'7'` only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
45188 decimal digits and letters `'a'` (or `'A'`) to `'f'` (or `'F'`) with values 10 to 15 respectively.

45189 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
45190 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
45191 by a `'+'` or `'−'` sign. The letters from `'a'` (or `'A'`) to `'z'` (or `'Z'`) inclusive are ascribed the
45192 values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the
45193 value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and
45194 digits, following the sign if present.

45195 The subject sequence is defined as the longest initial subsequence of the input string, starting
45196 with the first non-white-space character that is of the expected form. The subject sequence shall |
45197 contain no characters if the input string is empty or consists entirely of white-space characters, |
45198 or if the first non-white-space character is other than a sign or a permissible letter or digit.

45199 If the subject sequence has the expected form and the value of *base* is 0, the sequence of |
45200 characters starting with the first digit shall be interpreted as an integer constant. If the subject |
45201 sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the |
45202 base for conversion, ascribing to each letter its value as given above. If the subject sequence |
45203 begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to |
45204 the final string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null |
45205 pointer.

45206 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
45207 accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX The *strtol*( ) function shall not change the setting of *errno* if successful.

Since 0, {LONG_MIN} or {LLONG_MIN}, and {LONG_MAX} or {LLONG_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set *errno* to 0, then call *strtol*( ) or *strtoll*( ), then check *errno*.

RETURN VALUE

Upon successful completion, these functions shall return the converted value, if any. If no CX conversion could be performed, 0 shall be returned and *errno* may be set to [EINVAL].

If the correct value is outside the range of representable values, {LONG_MIN}, {LONG_MAX}, {LLONG_MIN} or {LLONG_MAX} shall be returned (according to the sign of the value), and *errno* set to [ERANGE].

ERRORS

These functions shall fail if:

[ERANGE] The value to be returned is not representable.

These functions may fail if:

CX [EINVAL] The value of *base* is not supported.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalpha( ), *scanf*( ), *strtod*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

- The *strtol*( ) prototype is updated.
- The *strtoll*( ) function is added.

45249 **NAME**

45250 strtold — convert string to a double-precision number

45251 **SYNOPSIS**

45252 `#include <stdlib.h>`

45253 `long double strtold(const char *restrict nptr, char **restrict endptr);`

45254 **DESCRIPTION**

45255 Refer to *strtod*( ).

45256 **NAME**

45257 strtoll — convert string to a long integer

45258 **SYNOPSIS**

45259 `#include <stdlib.h>`

45260 `long long strtoll(const char *restrict str, char **restrict endptr,`
45261 `int base);`

45262 **DESCRIPTION**

45263 Refer to *strtol*( ).

NAME

strtoul, strtoull — convert string to an unsigned long

SYNOPSIS

```
#include <stdlib.h>

unsigned long strtoul(const char *restrict str,
    char **restrict endptr, int base);
unsigned long long strtoull(const char *restrict str,
    char **restrict endptr, int base);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall convert the initial portion of the string pointed to by *str* to a type **unsigned long** and **unsigned long long** representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace*( ))
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string

Then they shall attempt to convert the subject sequence to an unsigned integer, and return the result.

If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a '+' or '−' sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7' only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a '+' or '−' sign. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character that is of the expected form. The subject sequence shall contain no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of characters starting with the first digit shall be interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to the final string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

45309 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
45310 accepted.

45311 If the subject sequence is empty or does not have the expected form, no conversion shall be |
45312 performed; the value of *str* shall be stored in the object pointed to by *endptr*, provided that *endptr* |
45313 is not a null pointer.

45314 CX The *strtoul*( ) function shall not change the setting of *errno* if successful.

45315 Since 0, {ULONG_MAX}, and {ULLONG_MAX} are returned on error and are also valid returns |
45316 on success, an application wishing to check for error situations should set *errno* to 0, then call
45317 *strtoul*( ) or *strtoull*( ), then check *errno*.

45318 **RETURN VALUE**

45319 Upon successful completion, these functions shall return the converted value, if any. If no
45320 CX conversion could be performed, 0 shall be returned and *errno* may be set to [EINVAL]. If the
45321 correct value is outside the range of representable values, {ULONG_MAX} or {ULLONG_MAX}
45322 shall be returned and *errno* set to [ERANGE].

45323 **ERRORS**

45324 These functions shall fail if:

45325 CX [EINVAL] The value of *base* is not supported.

45326 [ERANGE] The value to be returned is not representable.

45327 These functions may fail if:

45328 CX [EINVAL] No conversion could be performed.

45329 **EXAMPLES**

45330 None.

45331 **APPLICATION USAGE**

45332 None.

45333 **RATIONALE**

45334 None.

45335 **FUTURE DIRECTIONS**

45336 None.

45337 **SEE ALSO**

45338 *isalpha*( ), *scanf*( ), *strtod*( ), *strtol*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
45339 **<stdlib.h>**

45340 **CHANGE HISTORY**

45341 First released in Issue 4. Derived from the ANSI C standard.

45342 **Issue 5**

45343 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

45344 **Issue 6**

45345 Extensions beyond the ISO C standard are now marked.

45346 The following new requirements on POSIX implementations derive from alignment with the
45347 Single UNIX Specification:

45348 - The [EINVAL] error condition is added for when the value of *base* is not supported.

45349 In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
45350 added if no conversion could be performed.

The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

- The *strtoul*( ) prototype is updated.
- The *strtoull*( ) function is added.

45354 **NAME**

45355 strtoumax — convert string to integer type

45356 **SYNOPSIS**

45357 `#include <inttypes.h>`

45358 `uintmax_t strtoumax(const char *restrict` *`nptr`*`, char **restrict` *`endptr`*`,`
45359 `    int` *`base`*`);`

45360 **DESCRIPTION**

45361 Refer to *strtoimax*( ).

45362 **NAME**

45363 strxfrm — string transformation

45364 **SYNOPSIS**

45365
```
#include <string.h>
```

45366
```
size_t strxfrm(char *restrict s1, const char *restrict s2, size_t n);
```

45367 **DESCRIPTION**

45368 CX The functionality described on this reference page is aligned with the ISO C standard. Any
45369 conflict between the requirements described here and the ISO C standard is unintentional. This
45370 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45371 The *strxfrm*( ) function shall transform the string pointed to by *s2* and place the resulting string
45372 into the array pointed to by *s1*. The transformation is such that if *strcmp*( ) is applied to two
45373 transformed strings, it shall return a value greater than, equal to, or less than 0, corresponding to
45374 the result of *strcoll*( ) applied to the same two original strings. No more than *n* bytes are placed
45375 into the resulting array pointed to by *s1*, including the terminating null byte. If *n* is 0, *s1* is
45376 permitted to be a null pointer. If copying takes place between objects that overlap, the behavior
45377 is undefined.

45378 CX The *strxfrm*( ) function shall not change the setting of *errno* if successful.

45379 Since no return value is reserved to indicate an error, an application wishing to check for error |
45380 situations should set *errno* to 0, then call *strxfrm*( ), then check *errno*. |

45381 **RETURN VALUE**

45382 Upon successful completion, *strxfrm*( ) shall return the length of the transformed string (not
45383 including the terminating null byte). If the value returned is *n* or more, the contents of the array
45384 pointed to by *s1* are unspecified. |

45385 CX On error, *strxfrm*( ) may set *errno* but no return value is reserved to indicate an error.

45386 **ERRORS**

45387 The *strxfrm*( ) function may fail if:

45388 CX [EINVAL] The string pointed to by the *s2* argument contains characters outside the
45389 domain of the collating sequence.

45390 **EXAMPLES**

45391 None.

45392 **APPLICATION USAGE**

45393 The transformation function is such that two transformed strings can be ordered by *strcmp*( ) as
45394 appropriate to collating sequence information in the program's locale (category *LC_COLLATE*).

45395 The fact that when *n* is 0 *s1* is permitted to be a null pointer is useful to determine the size of the
45396 *s1* array prior to making the transformation.

45397 **RATIONALE**

45398 None.

45399 **FUTURE DIRECTIONS**

45400 None.

45401 **SEE ALSO**

45402 *strcmp*( ), *strcoll*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

45403 **CHANGE HISTORY**

45404 First released in Issue 3.

45405 Entry included for alignment with the ISO C standard.

45406 **Issue 5**

45407 The DESCRIPTION is updated to indicate that *errno* does not changed if the function is
45408 successful.

45409 **Issue 6**

45410 Extensions beyond the ISO C standard are now marked.

45411 The following new requirements on POSIX implementations derive from alignment with the
45412 Single UNIX Specification:

45413 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
45414 added if no conversion could be performed.

45415 The *strxfrm*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

swab — swap bytes

SYNOPSIS

XSI
```
#include <unistd.h>

void swab(const void *restrict src, void *restrict dest,
    ssize_t nbytes);
```

DESCRIPTION

The *swab*( ) function shall copy *nbytes* bytes, which are pointed to by *src*, to the object pointed to by *dest*, exchanging adjacent bytes. The *nbytes* argument should be even. If *nbytes* is odd, *swab*( ) copies and exchanges *nbytes*−1 bytes and the disposition of the last byte is unspecified. If copying takes place between objects that overlap, the behavior is undefined. If *nbytes* is negative, *swab*( ) does nothing.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The **restrict** keyword is added to the *swab*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

45448 **NAME**

45449 swapcontext — swap user context

45450 **SYNOPSIS**

45451 XSI
```
#include <ucontext.h>

int swapcontext(ucontext_t *restrict oucp,
    const ucontext_t *restrict ucp);
```
45454

45455 **DESCRIPTION**

45456 Refer to *makecontext*( ).

45457 **NAME**

45458 swprintf — print formatted wide-character output

45459 **SYNOPSIS**

45460 ```
#include <stdio.h>
```

45461 ```
#include <wchar.h>
```

45462 ```
int swprintf(wchar_t *ws, size_t n, const wchar_t *format, ...);
```

45463 **DESCRIPTION**

45464 Refer to *fwprintf*( ).
```

45465 **NAME**

45466 swscanf — convert formatted wide-character input

45467 **SYNOPSIS**

45468 `#include <stdio.h>`

45469 `#include <wchar.h>`

45470 `int swscanf(const wchar_t *restrict ws,` |

45471 `    const wchar_t *restrict format, ... );` |

45472 **DESCRIPTION** |

45473 Refer to *fwscanf*( ).

45474 **NAME**

45475 symlink — make symbolic link to a file

45476 **SYNOPSIS**

45477 `#include <unistd.h>`

45478 `int symlink(const char *path1, const char *path2);`

45479 **DESCRIPTION**

45480 The *symlink*( ) function shall create a symbolic link called *path2* that contains the string pointed
45481 to by *path1* (*path2* is the name of the symbolic link created, *path1* is the string contained in the
45482 symbolic link).

45483 The string pointed to by *path1* shall be treated only as a character string and shall not be
45484 validated as a pathname. |

45485 If the *symlink*( ) function fails for any reason other than [EIO], any file named by *path2* shall be
45486 unaffected.

45487 **RETURN VALUE**

45488 Upon successful completion, *symlink*( ) shall return 0; otherwise, it shall return –1 and set *errno* to
45489 indicate the error.

45490 **ERRORS**

45491 The *symlink*( ) function shall fail if:

45492 [EACCES] Write permission is denied in the directory where the symbolic link is being
45493 created, or search permission is denied for a component of the path prefix of
45494 *path2*.

45495 [EEXIST] The *path2* argument names an existing file or symbolic link.

45496 [EIO] An I/O error occurs while reading from or writing to the file system.

45497 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path2*
45498 argument.

45499 [ENAMETOOLONG]
45500 The length of the *path2* argument exceeds {PATH_MAX} or a pathname |
45501 component is longer than {NAME_MAX} or the length of the *path1* argument |
45502 is longer than {SYMLINK_MAX}.

45503 [ENOENT] A component of *path2* does not name an existing file or *path2* is an empty
45504 string.

45505 [ENOSPC] The directory in which the entry for the new symbolic link is being placed
45506 cannot be extended because no space is left on the file system containing the
45507 directory, or the new symbolic link cannot be created because no space is left
45508 on the file system which shall contain the link, or the file system is out of file-
45509 allocation resources.

45510 [ENOTDIR] A component of the path prefix of *path2* is not a directory.

45511 [EROFS] The new symbolic link would reside on a read-only file system.

45512 The *symlink*( ) function may fail if:

45513 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
45514 resolution of the *path2* argument.

45515 [ENAMETOOLONG]
45516 As a result of encountering a symbolic link in resolution of the *path2* |

argument, the length of the substituted pathname string exceeded {PATH_MAX} bytes (including the terminating null byte), or the length of the string pointed to by *path1* exceeded {SYMLINK_MAX}.

EXAMPLES

None.

APPLICATION USAGE

Like a hard link, a symbolic link allows a file to have multiple logical names. The presence of a hard link guarantees the existence of a file, even after the original name has been removed. A symbolic link provides no such assurance; in fact, the file named by the *path1* argument need not exist when the link is created. A symbolic link can cross file system boundaries.

Normal permission checks are made on each component of the symbolic link pathname during its resolution.

RATIONALE

Since IEEE Std 1003.1-200x does not require any association of file times with symbolic links, there is no requirement that file times be updated by *symlink*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

lchown( ), *link*( ), *lstat*( ), *open*( ), *readlink*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The following changes were made to align with the IEEE P1003.1a draft standard:

- The DESCRIPTION text is updated.
- The [ELOOP] optional error condition is added.

45545 **NAME**

45546 sync — schedule file system updates

45547 **SYNOPSIS**

45548 XSI
```
#include <unistd.h>
```

45549
```
void sync(void);
```

45550

45551 **DESCRIPTION**

45552 The *sync*( ) function shall cause all information in memory that updates file systems to be
45553 scheduled for writing out to all file systems.

45554 The writing, although scheduled, is not necessarily complete upon return from *sync*( ).

45555 **RETURN VALUE**

45556 The *sync*( ) function shall not return a value.

45557 **ERRORS**

45558 No errors are defined.

45559 **EXAMPLES**

45560 None.

45561 **APPLICATION USAGE**

45562 None.

45563 **RATIONALE**

45564 None.

45565 **FUTURE DIRECTIONS**

45566 None.

45567 **SEE ALSO**

45568 *fsync*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

45569 **CHANGE HISTORY**

45570 First released in Issue 4, Version 2.

45571 **Issue 5**

45572 Moved from X/OPEN UNIX extension to BASE.

NAME

sysconf — get configurable system variables

SYNOPSIS

```
#include <unistd.h>

long sysconf(int name);
```

DESCRIPTION

The *sysconf*( ) function provides a method for the application to determine the current value of a configurable system limit or option (*variable*). Support for some system variables is dependent on implementation options (as indicated by the margin codes in the following table). Where an implementation option is not supported, the variable need not be supported.

The *name* argument represents the system variable to be queried. The following table lists the minimal set of system variables from **<limits.h>** or **<unistd.h>** that can be returned by *sysconf*( ), and the symbolic constants, defined in **<unistd.h>** that are the corresponding values used for *name*. Support for some configuration variables is dependent on implementation options (see shading and margin codes in the table below). Where an implementation option is not supported, the variable need not be supported.

	Variable	Value of Name
AIO	{AIO_LISTIO_MAX}	_SC_AIO_LISTIO_MAX
	{AIO_MAX}	_SC_AIO_MAX
	{AIO_PRIO_DELTA_MAX}	_SC_AIO_PRIO_DELTA_MAX
	{ARG_MAX}	_SC_ARG_MAX
XSI	{ATEXIT_MAX}	_SC_ATEXIT_MAX
	{BC_BASE_MAX}	_SC_BC_BASE_MAX
	{BC_DIM_MAX}	_SC_BC_DIM_MAX
	{BC_SCALE_MAX}	_SC_BC_SCALE_MAX
	{BC_STRING_MAX}	_SC_BC_STRING_MAX
	{CHILD_MAX}	_SC_CHILD_MAX
	Clock ticks/second	_SC_CLK_TCK
	{COLL_WEIGHTS_MAX}	_SC_COLL_WEIGHTS_MAX
XSI	{DELAYTIMER_MAX}	_SC_DELAYTIMER_MAX
	{EXPR_NEST_MAX}	_SC_EXPR_NEST_MAX
	{HOST_NAME_MAX}	_SC_HOST_NAME_MAX
XSI	{IOV_MAX}	_SC_IOV_MAX
	{LINE_MAX}	_SC_LINE_MAX
	{LOGIN_NAME_MAX}	_SC_LOGIN_NAME_MAX
	{NGROUPS_MAX}	_SC_NGROUPS_MAX
TSF	Maximum size of *getgrgid_r*( ) and *getgrnam_r*( ) data buffers	_SC_GETGR_R_SIZE_MAX
	Maximum size of *getpwuid_r*( ) and *getpwnam_r*( ) data buffers	_SC_GETPW_R_SIZE_MAX
MSG	{MQ_OPEN_MAX}	_SC_MQ_OPEN_MAX
	{MQ_PRIO_MAX}	_SC_MQ_PRIO_MAX
	{OPEN_MAX}	_SC_OPEN_MAX

	Variable	Value of Name
ADV	_POSIX_ADVISORY_INFO	_SC_ADVISORY_INFO
BAR	_POSIX_BARRIERS	_SC_BARRIERS
AIO	_POSIX_ASYNCHRONOUS_IO	_SC_ASYNCHRONOUS_IO
	_POSIX_BASE	_SC_BASE
	_POSIX_C_LANG_SUPPORT	_SC_C_LANG_SUPPORT
	_POSIX_C_LANG_SUPPORT_R	_SC_C_LANG_SUPPORT_R
CS	_POSIX_CLOCK_SELECTION	_SC_CLOCK_SELECTION
CPT	_POSIX_CPUTIME	_SC_CPUTIME
	_POSIX_DEVICE_IO	_SC_DEVICE_IO
	_POSIX_DEVICE_SPECIFIC	_SC_DEVICE_SPECIFIC
	_POSIX_DEVICE_SPECIFIC_R	_SC_DEVICE_SPECIFIC_R
	_POSIX_FD_MGMT	_SC_FD_MGMT
	_POSIX_FIFO	_SC_FIFO
	_POSIX_FILE_ATTRIBUTES	_SC_FILE_ATTRIBUTES
	_POSIX_FILE_LOCKING	_SC_FILE_LOCKING
	_POSIX_FILE_SYSTEM	_SC_FILE_SYSTEM
FSC	_POSIX_FSYNC	_SC_FSYNC
	_POSIX_JOB_CONTROL	_SC_JOB_CONTROL
MF	_POSIX_MAPPED_FILES	_SC_MAPPED_FILES
ML	_POSIX_MEMLOCK	_SC_MEMLOCK
MLR	_POSIX_MEMLOCK_RANGE	_SC_MEMLOCK_RANGE
MPR	_POSIX_MEMORY_PROTECTION	_SC_MEMORY_PROTECTION
MSG	_POSIX_MESSAGE_PASSING	_SC_MESSAGE_PASSING
MON	_POSIX_MONOTONIC_CLOCK	_SC_MONOTONIC_CLOCK
	_POSIX_MULTI_PROCESS	_SC_MULTI_PROCESS
	_POSIX_NETWORKING	_SC_NETWORKING
	_POSIX_PIPE	_SC_PIPE
PIO	_POSIX_PRIORITIZED_IO	_SC_PRIORITIZED_IO
PS	_POSIX_PRIORITY_SCHEDULING	_SC_PRIORITY_SCHEDULING
THR	_POSIX_READER_WRITER_LOCKS	_SC_READER_WRITER_LOCKS
RTS	_POSIX_REALTIME_SIGNALS	_SC_REALTIME_SIGNALS
	_POSIX_REGEXP	_SC_REGEXP
	_POSIX_SAVED_IDS	_SC_SAVED_IDS
SEM	_POSIX_SEMAPHORES	_SC_SEMAPHORES
SHM	_POSIX_SHARED_MEMORY_OBJECTS	_SC_SHARED_MEMORY_OBJECTS
	_POSIX_SHELL	_SC_SHELL
	_POSIX_SIGNALS	_SC_SIGNALS
	_POSIX_SINGLE_PROCESS	_SC_SINGLE_PROCESS
SPN	_POSIX_SPAWN	_SC_SPAWN
SPI	_POSIX_SPIN_LOCKS	_SC_SPIN_LOCKS
SS	_POSIX_SPORADIC_SERVER	_SC_SPORADIC_SERVER
SIO	_POSIX_SYNCHRONIZED_IO	_SC_SYNCHRONIZED_IO
	_POSIX_SYSTEM_DATABASE	_SC_SYSTEM_DATABASE
	_POSIX_SYSTEM_DATABASE_R	_SC_SYSTEM_DATABASE_R

	Variable	Value of Name
TSA	_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR
TSS	_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE
TCT	_POSIX_THREAD_CPUTIME	_SC_THREAD_CPUTIME
TPI	_POSIX_THREAD_PRIO_INHERIT	_SC_THREAD_PRIO_INHERIT
TPP	_POSIX_THREAD_PRIO_PROTECT	_SC_THREAD_PRIO_PROTECT
TPS	_POSIX_THREAD_PRIORITY_SCHEDULING	_SC_THREAD_PRIORITY_SCHEDULING
TSH	_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED
TSF	_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS
TSP	_POSIX_THREAD_SPORADIC_SERVER	_SC_THREAD_SPORADIC_SERVER
THR	_POSIX_THREADS	_SC_THREADS
TMO	_POSIX_TIMEOUTS	_SC_TIMEOUTS
TMR	_POSIX_TIMERS	_SC_TIMERS
TRC	_POSIX_TRACE	_SC_TRACE
TEF	_POSIX_TRACE_EVENT_FILTER	_SC_TRACE_EVENT_FILTER
TRI	_POSIX_TRACE_INHERIT	_SC_TRACE_INHERIT
TRL	_POSIX_TRACE_LOG	_SC_TRACE_LOG
TYM	_POSIX_TYPED_MEMORY_OBJECTS	_SC_TYPED_MEMORY_OBJECTS
	_POSIX_USER_GROUPS	_SC_USER_GROUPS
	_POSIX_USER_GROUPS_R	_SC_USER_GROUPS_R
	_POSIX_VERSION	_SC_VERSION
	_POSIX_V6_ILP32_OFF32	_SC_V6_ILP32_OFF32
	_POSIX_V6_ILP32_OFFBIG	_SC_V6_ILP32_OFFBIG
	_POSIX_V6_LP64_OFF64	_SC_V6_LP64_OFF64
	_POSIX_V6_LPBIG_OFFBIG	_SC_V6_LPBIG_OFFBIG
	_POSIX2_C_BIND	_SC_2_C_BIND
	_POSIX2_C_DEV	_SC_2_C_DEV
	_POSIX2_C_VERSION	_SC_2_C_VERSION
	_POSIX2_CHAR_TERM	_SC_2_CHAR_TERM
	_POSIX2_FORT_DEV	_SC_2_FORT_DEV
	_POSIX2_FORT_RUN	_SC_2_FORT_RUN
	_POSIX2_LOCALEDEF	_SC_2_LOCALEDEF
BE	_POSIX2_PBS	_SC_2_PBS
	_POSIX2_PBS_ACCOUNTING	_SC_2_PBS_ACCOUNTING
	_POSIX2_PBS_LOCATE	_SC_2_PBS_LOCATE
	_POSIX2_PBS_MESSAGE	_SC_2_PBS_MESSAGE
	_POSIX2_PBS_TRACK	_SC_2_PBS_TRACK
	_POSIX2_SW_DEV	_SC_2_SW_DEV
	_POSIX2_UPE	_SC_2_UPE
	_POSIX2_VERSION	_SC_2_VERSION
	_REGEX_VERSION	_SC_REGEX_VERSION
XSI	{PAGE_SIZE}	_SC_PAGE_SIZE
	{PAGESIZE}	_SC_PAGESIZE

	Variable	Value of Name
THR	{PTHREAD_DESTRUCTOR_ITERATIONS}	_SC_THREAD_DESTRUCTOR_ITERATIONS
	{PTHREAD_KEYS_MAX}	_SC_THREAD_KEYS_MAX
	{PTHREAD_STACK_MIN}	_SC_THREAD_STACK_MIN
	{PTHREAD_THREADS_MAX}	_SC_THREAD_THREADS_MAX
	{RE_DUP_MAX}	_SC_RE_DUP_MAX
RTS	{RTSIG_MAX}	_SC_RTSIG_MAX
SEM	{SEM_NSEMS_MAX}	_SC_SEM_NSEMS_MAX
	{SEM_VALUE_MAX}	_SC_SEM_VALUE_MAX
RTS	{SIGQUEUE_MAX}	_SC_SIGQUEUE_MAX
	{STREAM_MAX}	_SC_STREAM_MAX
	{SYMLOOP_MAX}	_SC_SYMLOOP_MAX
TMR	{TIMER_MAX}	_SC_TIMER_MAX
	{TTY_NAME_MAX}	_SC_TTY_NAME_MAX
	{TZNAME_MAX}	_SC_TZNAME_MAX
XSI	_XBS5_ILP32_OFF32 (**LEGACY**)	_SC_XBS5_ILP32_OFF32 (**LEGACY**)
	_XBS5_ILP32_OFFBIG (**LEGACY**)	_SC_XBS5_ILP32_OFFBIG (**LEGACY**)
	_XBS5_LP64_OFF64 (**LEGACY**)	_SC_XBS5_LP64_OFF64 (**LEGACY**)
	_XBS5_LPBIG_OFFBIG (**LEGACY**)	_SC_XBS5_LPBIG_OFFBIG (**LEGACY**)
	_XOPEN_CRYPT	_SC_XOPEN_CRYPT
	_XOPEN_ENH_I18N	_SC_XOPEN_ENH_I18N
	_XOPEN_LEGACY	_SC_XOPEN_LEGACY
	_XOPEN_REALTIME	_SC_XOPEN_REALTIME
	_XOPEN_REALTIME_THREADS	_SC_XOPEN_REALTIME_THREADS
	_XOPEN_SHM	_SC_XOPEN_SHM
	_XOPEN_UNIX	_SC_XOPEN_UNIX
	_XOPEN_VERSION	_SC_XOPEN_VERSION
	_XOPEN_XCU_VERSION	_SC_XOPEN_XCU_VERSION

RETURN VALUE

If *name* is an invalid value, *sysconf*( ) shall return –1 and set *errno* to indicate the error. If the variable corresponding to *name* has no limit, *sysconf*( ) shall return –1 without changing the value of *errno*. Note that indefinite limits do not imply infinite limits; see **<limits.h>**.

Otherwise, *sysconf*( ) shall return the current variable value on the system. The value returned shall not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's **<limits.h>** or **<unistd.h>**. The value shall not change during the lifetime of the calling process.

ERRORS

The *sysconf*( ) function shall fail if:

[EINVAL] The value of the *name* argument is invalid.

EXAMPLES

None.

APPLICATION USAGE

As –1 is a permissible return value in a successful situation, an application wishing to check for error situations should set *errno* to 0, then call *sysconf*( ), and, if it returns –1, check to see if *errno* is non-zero.

If the value of *sysconf*(_SC_2_VERSION) is not equal to the value of the _POSIX2_VERSION symbolic constant, the utilities available via *system*( ) or *popen*( ) might not behave as described in the Shell and Utilities volume of IEEE Std 1003.1-200x. This would mean that the application is

not running in an environment that conforms to the Shell and Utilities volume of IEEE Std 1003.1-200x. Some applications might be able to deal with this, others might not. However, the functions defined in this volume of IEEE Std 1003.1-200x continue to operate as specified, even if: *sysconf*(_SC_2_VERSION) reports that the utilities no longer perform as specified.

RATIONALE

This functionality was added in response to requirements of application developers and of system vendors who deal with many international system configurations. It is closely related to *pathconf*( ) and *fpathconf*( ).

Although a conforming application can run on all systems by never demanding more resources than the minimum values published in this volume of IEEE Std 1003.1-200x, it is useful for that application to be able to use the actual value for the quantity of a resource available on any given system. To do this, the application makes use of the value of a symbolic constant in **<limits.h>** or **<unistd.h>**.

However, once compiled, the application must still be able to cope if the amount of resource available is increased. To that end, an application may need a means of determining the quantity of a resource, or the presence of an option, at execution time.

Two examples are offered:

1. Applications may wish to act differently on systems with or without job control. Applications vendors who wish to distribute only a single binary package to all instances of a computer architecture would be forced to assume job control is never available if it were to rely solely on the **<unistd.h>** value published in this volume of IEEE Std 1003.1-200x.

2. International applications vendors occasionally require knowledge of the number of clock ticks per second. Without these facilities, they would be required to either distribute their applications partially in source form or to have 50Hz and 60Hz versions for the various countries in which they operate.

It is the knowledge that many applications are actually distributed widely in executable form that leads to this facility. If limited to the most restrictive values in the headers, such applications would have to be prepared to accept the most limited environments offered by the smallest microcomputers. Although this is entirely portable, there was a consensus that they should be able to take advantage of the facilities offered by large systems, without the restrictions associated with source and object distributions.

During the discussions of this feature, it was pointed out that it is almost always possible for an application to discern what a value might be at runtime by suitably testing the various functions themselves. And, in any event, it could always be written to adequately deal with error returns from the various functions. In the end, it was felt that this imposed an unreasonable level of complication and sophistication on the application writer.

This runtime facility is not meant to provide ever-changing values that applications have to check multiple times. The values are seen as changing no more frequently than once per system initialization, such as by a system administrator or operator with an automatic configuration program. This volume of IEEE Std 1003.1-200x specifies that they shall not change within the lifetime of the process.

Some values apply to the system overall and others vary at the file system or directory level. The latter are described in *pathconf*( ).

Note that all values returned must be expressible as integers. String values were considered, but the additional flexibility of this approach was rejected due to its added complexity of

45803 implementation and use.

45804 Some values, such as {PATH_MAX}, are sometimes so large that they must not be used to, say,
45805 allocate arrays. The *sysconf*( ) function returns a negative value to show that this symbolic
45806 constant is not even defined in this case.

45807 Similar to *pathconf*( ), this permits the implementation not to have a limit. When one resource is
45808 infinite, returning an error indicating that some other resource limit has been reached is
45809 conforming behavior.

45810 **FUTURE DIRECTIONS**

45811 None.

45812 **SEE ALSO**

45813 *confstr*( ), *pathconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**,
45814 **<unistd.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *getconf*

45815 **CHANGE HISTORY**

45816 First released in Issue 3.

45817 Entry included for alignment with the POSIX.1-1988 standard.

45818 **Issue 5**

45819 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
45820 Threads Extension.

45821 The _XBS_ variables and name values are added to the table of system variables in the
45822 DESCRIPTION. These are all marked EX.

45823 **Issue 6**

45824 The symbol CLK_TCK is obsolescent and removed. It is replaced with the phrase ‘‘clock ticks
45825 per second’’.

45826 The symbol {PASS_MAX} is removed.

45827 The following changes were made to align with the IEEE P1003.1a draft standard:

45828 • Table entries added for the following variables: _SC_REGEXP, _SC_SHELL,
45829 _SC_REGEX_VERSION, _SC_SYMLOOP_MAX.

45830 The following *sysconf*( ) variables and their associated names are added for alignment with
45831 IEEE Std 1003.1d-1999:

45832 _POSIX_ADVISORY_INFO
45833 _POSIX_CPUTIME
45834 _POSIX_SPAWN
45835 _POSIX_SPORADIC_SERVER
45836 _POSIX_THREAD_CPUTIME
45837 _POSIX_THREAD_SPORADIC_SERVER
45838 _POSIX_TIMEOUTS

45839 The following changes are made to the DESCRIPTION for alignment with IEEE Std 1003.1j-2000:

45840 • A statement expressing the dependency of support for some system variables on
45841 implementation options is added.

45842 • The following system variables are added:

45843 _POSIX_BARRIERS
45844 _POSIX_CLOCK_SELECTION
45845 _POSIX_MONOTONIC_CLOCK
45846 _POSIX_READER_WRITER_LOCKS
45847 _POSIX_SPIN_LOCKS
45848 _POSIX_TYPED_MEMORY_OBJECTS

45849 The following system variables are added for alignment with IEEE Std 1003.2d-1994:

45850 _POSIX2_PBS
45851 _POSIX2_PBS_ACCOUNTING
45852 _POSIX2_PBS_LOCATE
45853 _POSIX2_PBS_MESSAGE
45854 _POSIX2_PBS_TRACK

45855 The following *sysconf*() variables and their associated names are added for alignment with
45856 IEEE Std 1003.1q-2000:

45857 _POSIX_TRACE
45858 _POSIX_TRACE_EVENT_FILTER
45859 _POSIX_TRACE_INHERIT
45860 _POSIX_TRACE_LOG

45861 The macros associated with the *c89* programming models are marked LEGACY, and new
45862 equivalent macros associated with *c99* are introduced.

45863 **NAME**

45864 syslog — log a message

45865 **SYNOPSIS**

45866 XSI
```
#include <syslog.h>
```

45867
```
void syslog(int priority, const char *message, ... /* argument */);
```

45868

45869 **DESCRIPTION**

45870 Refer to *closelog*( ).

45871 **NAME**

45872 system — issue a command

45873 **SYNOPSIS**

45874 `#include <stdlib.h>`

45875 `int system(const char *command);`

45876 **DESCRIPTION**

45877 CX The functionality described on this reference page is aligned with the ISO C standard. Any
45878 conflict between the requirements described here and the ISO C standard is unintentional. This
45879 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45880 If *command* is a null pointer, the *system*( ) function shall determine whether the host environment
45881 has a command processor. If *command* is not a null pointer, the *system*( ) function shall pass the
45882 string pointed to by *command* to that command processor to be executed in an implementation-
45883 defined manner; this might then cause the program calling *system*( ) to behave in a non-
45884 conforming manner or to terminate.

45885 CX The environment of the executed command shall be as if a child process were created using
45886 *fork*( ), and the child process invoked the *sh* utility using *execl*( ) as follows:

45887 `execl(<shell path>, "sh", "-c", command, (char *)0);`

45888 where <*shell path*> is an unspecified pathname for the *sh* utility.

45889 The *system*( ) function shall ignore the SIGINT and SIGQUIT signals, and shall block the
45890 SIGCHLD signal, while waiting for the command to terminate. If this might cause the
45891 application to miss a signal that would have killed it, then the application should examine the
45892 return value from *system*( ) and take whatever action is appropriate to the application if the
45893 command terminated due to receipt of a signal.

45894 The *system*( ) function shall not affect the termination status of any child of the calling processes
45895 other than the process or processes it itself creates.

45896 The *system*( ) function shall not return until the child process has terminated.

45897 **RETURN VALUE**

45898 If *command* is a null pointer, *system*( ) shall return non-zero to indicate that a command processor
45899 CX is available, or zero if none is available. The *system*( ) function shall always return non-zero when
45900 *command* is NULL.

45901 CX If *command* is not a null pointer, *system*( ) shall return the termination status of the command
45902 language interpreter in the format specified by *waitpid*( ). The termination status shall be as
45903 defined for the *sh* utility; otherwise, the termination status is unspecified. If some error prevents
45904 the command language interpreter from executing after the child process is created, the return
45905 value from *system*( ) shall be as if the command language interpreter had terminated using
45906 *exit*(127) or *_exit*(127). If a child process cannot be created, or if the termination status for the
45907 command language interpreter cannot be obtained, *system*( ) shall return –1 and set *errno* to
45908 indicate the error.

45909 **ERRORS**

45910 CX The *system*( ) function may set *errno* values as described by *fork*( ).

45911 In addition, *system*( ) may fail if:

45912 CX [ECHILD] The status of the child process created by *system*( ) is no longer available.

45913 **EXAMPLES**

45914 None.

45915 **APPLICATION USAGE**

45916 If the return value of *system*( ) is not –1, its value can be decoded through the use of the macros
45917 described in **<sys/wait.h>**. For convenience, these macros are also provided in **<stdlib.h>**.

45918 Note that, while *system*( ) must ignore SIGINT and SIGQUIT and block SIGCHLD while waiting |
45919 for the child to terminate, the handling of signals in the executed command is as specified by
45920 *fork*( ) and *exec*. For example, if SIGINT is being caught or is set to SIG_DFL when *system*( ) is
45921 called, then the child is started with SIGINT handling set to SIG_DFL.

45922 Ignoring SIGINT and SIGQUIT in the parent process prevents coordination problems (two
45923 processes reading from the same terminal, for example) when the executed command ignores or
45924 catches one of the signals. It is also usually the correct action when the user has given a
45925 command to the application to be executed synchronously (as in the '!' command in many
45926 interactive applications). In either case, the signal should be delivered only to the child process,
45927 not to the application itself. There is one situation where ignoring the signals might have less
45928 than the desired effect. This is when the application uses *system*( ) to perform some task invisible
45929 to the user. If the user typed the interrupt character ("^C", for example) while *system*( ) is being
45930 used in this way, one would expect the application to be killed, but only the executed command
45931 is killed. Applications that use *system*( ) in this way should carefully check the return status from
45932 *system*( ) to see if the executed command was successful, and should take appropriate action
45933 when the command fails.

45934 Blocking SIGCHLD while waiting for the child to terminate prevents the application from
45935 catching the signal and obtaining status from *system*( )'s child process before *system*( ) can get the
45936 status itself.

45937 The context in which the utility is ultimately executed may differ from that in which *system*( )
45938 was called. For example, file descriptors that have the FD_CLOEXEC flag set are closed, and the
45939 process ID and parent process ID are different. Also, if the executed utility changes its
45940 environment variables or its current working directory, that change is not reflected in the caller's
45941 context.

45942 There is no defined way for an application to find the specific path for the shell. However,
45943 *confstr*( ) can provide a value for *PATH* that is guaranteed to find the *sh* utility.

45944 **RATIONALE**

45945 The *system*( ) function should not be used by programs that have set user (or group) ID
45946 privileges. The *fork*( ) and *exec* family of functions (except *execlp*( ) and *execvp*( )), should be used
45947 instead. This prevents any unforeseen manipulation of the environment of the user that could
45948 cause execution of commands not anticipated by the calling program.

45949 There are three levels of specification for the *system*( ) function. The ISO C standard gives the
45950 most basic. It requires that the function exists, and defines a way for an application to query
45951 whether a command language interpreter exists. It says nothing about the command language or
45952 the environment in which the command is interpreted.

45953 IEEE Std 1003.1-200x places additional restrictions on *system*( ). It requires that if there is a
45954 command language interpreter, the environment must be as specified by *fork*( ) and *exec*. This
45955 ensures, for example, that close-on-*exec* works, that file locks are not inherited, and that the
45956 process ID is different. It also specifies the return value from *system*( ) when the command line
45957 can be run, thus giving the application some information about the command's completion
45958 statu.

Finally, IEEE Std 1003.1-200x requires the command to be interpreted as in the shell command language defined in the Shell and Utilities volume of IEEE Std 1003.1-200x.

Note that, *system*(NULL) is required to return non-zero, indicating that there is a command language interpreter. At first glance, this would seem to conflict with the ISO C standard which allows *system*(NULL) to return zero. There is no conflict, however. A system must have a command language interpreter, and is non-conforming if none is present. It is therefore permissible for the *system*( ) function on such a system to implement the behavior specified by the ISO C standard as long as it is understood that the implementation does not conform to IEEE Std 1003.1-200x if *system*(NULL) returns zero.

It was explicitly decided that when *command* is NULL, *system*( ) should not be required to check to make sure that the command language interpreter actually exists with the correct mode, that there are enough processes to execute it, and so on. The call *system*(NULL) could, theoretically, check for such problems as too many existing child processes, and return zero. However, it would be inappropriate to return zero due to such a (presumably) transient condition. If some condition exists that is not under the control of this application and that would cause any *system*( ) call to fail, that system has been rendered non-conforming.

Early drafts required, or allowed, *system*( ) to return with *errno* set to [EINTR] if it was interrupted with a signal. This error return was removed, and a requirement that *system*( ) not return until the child has terminated was added. This means that if a *waitpid*( ) call in *system*( ) exits with *errno* set to [EINTR], *system*( ) must re-issue the *waitpid*( ). This change was made for two reasons:

1. There is no way for an application to clean up if *system*( ) returns [EINTR], short of calling *wait*( ), and that could have the undesirable effect of returning the status of children other than the one started by *system*( ).
2. While it might require a change in some historical implementations, those implementations already have to be changed because they use *wait*( ) instead of *waitpid*( ).

Note that if the application is catching SIGCHLD signals, it will receive such a signal before a successful *system*( ) call returns.

To conform to IEEE Std 1003.1-200x, *system*( ) must use *waitpid*( ), or some similar function, instead of *wait*( ).

The following code sample illustrates how *system*( ) might be implemented on an implementation conforming to IEEE Std 1003.1-200x.

```
#include <signal.h>
int system(const char *cmd)
{
    int stat;
    pid_t pid;
    struct sigaction sa, savintr, savequit;
    sigset_t saveblock;
    if (cmd == NULL)
        return(1);
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigemptyset(&savintr.sa_mask);
    sigemptyset(&savequit.sa_mask);
    sigaction(SIGINT, &sa, &savintr);
    sigaction(SIGQUIT, &sa, &savequit);
```

```
        sigaddset(&sa.sa_mask, SIGCHLD);
        sigprocmask(SIG_BLOCK, &sa.sa_mask, &saveblock);
        if ((pid = fork()) == 0) {
            sigaction(SIGINT, &savintr, (struct sigaction *)0);
            sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
            sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
            execl("/bin/sh", "sh", "-c", cmd, (char *)0);
            _exit(127);
        }
        if (pid == -1) {
            stat = -1; /* errno comes from fork() */
        } else {
            while (waitpid(pid, &stat, 0) == -1) {
                if (errno != EINTR){
                    stat = -1;
                    break;
                }
            }
        }
        sigaction(SIGINT, &savintr, (struct sigaction *)0);
        sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
        sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
        return(stat);
    }
```

Note that, while a particular implementation of *system*( ) (such as the one above) can assume a particular path for the shell, such a path is not necessarily valid on another system. The above example is not portable, and is not intended to be.

One reviewer suggested that an implementation of *system*( ) might want to use an environment variable such as *SHELL* to determine which command interpreter to use. The supposed implementation would use the default command interpreter if the one specified by the environment variable was not available. This would allow a user, when using an application that prompts for command lines to be processed using *system*( ), to specify a different command interpreter. Such an implementation is discouraged. If the alternate command interpreter did not follow the command line syntax specified in the Shell and Utilities volume of IEEE Std 1003.1-200x, then changing *SHELL* would render *system*( ) non-conforming. This would affect applications that expected the specified behavior from *system*( ), and since the Shell and Utilities volume of IEEE Std 1003.1-200x does not mention that *SHELL* affects *system*( ), the application would not know that it needed to unset *SHELL*.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *pipe*( ), *waitpid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<signal.h>**, **<stdlib.h>**, **<sys/wait.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *sh* |

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID. |

Issue 6

The following changes were made to align with the IEEE P1003.1a draft standard:

- The DESCRIPTION is adjusted to reflect the behavior on systems that do not support the Shell option.

NAME

tan, tanf, tanl — tangent function

SYNOPSIS

```
#include <math.h>

double tan(double x);
float tanf(float x);
long double tanl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the tangent of their argument *x*, measured in radians.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the tangent of *x*.

If the correct value would cause underflow, and is not representable, a range error may occur, and MX either 0.0 (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, *x* shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

XSI If the correct value would cause overflow, a range error shall occur and *tan*(), *tanf*(), and *tanl*() shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

ERRORS

These functions shall fail if:

MX	Domain Error	The value *x* is ±Inf.
		If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
XSI	Range Error	The result overflows
		If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

MX Range Error — The result underflows, or the value *x* is subnormal.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

Taking the Tangent of a 45-Degree Angle

```
#include <math.h>
...
double radians = 45.0 * M_PI / 180;
double result;
...
result = tan (radians);
```

APPLICATION USAGE

There are no known floating-point representations such that for a normal argument, *tan*(*x*) is either overflow or underflow.

These functions may lose accuracy when their argument is near a multiple of $\pi/2$ or is far from 0.0.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

atan( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes in previous issues.

Issue 6

The *tanf*( ) and *tanl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

46136 **NAME**

46137 tanf — tangent function

46138 **SYNOPSIS**

46139
```
#include <math.h>
```

46140
```
float tanf(float x);
```

46141 **DESCRIPTION**

46142 Refer to *tan*( ).

NAME

tanh, tanhf, tanhl — hyperbolic tangent functions

SYNOPSIS

```
#include <math.h>

double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the hyperbolic tangent of their argument *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the hyperbolic tangent of *x*.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, *x* shall be returned.

If *x* is ±Inf, ±1 shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

These functions may fail if:

MX

Range Error	The value of *x* is subnormal.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

atanh( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *tan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *tanhf*( ) and *tanhl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

46196 **NAME**

46197 tanl — tangent function

46198 **SYNOPSIS**

46199 `#include <math.h>`

46200 `long double tanl(long double x);`

46201 **DESCRIPTION**

46202 Refer to *tan*( ).

NAME

tcdrain — wait for transmission of output

SYNOPSIS

```
#include <termios.h>

int tcdrain(int fildes);
```

DESCRIPTION

The *tcdrain*( ) function shall block until all output written to the object referred to by *fildes* is transmitted. The *fildes* argument is an open file descriptor associated with a terminal.

Any attempts to use *tcdrain*( ) from a process which is a member of a background process group on a *fildes* associated with its controlling terminal, shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process shall be allowed to perform the operation, and no signal is sent.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcdrain*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINTR] A signal interrupted *tcdrain*( ).

[ENOTTY] The file associated with *fildes* is not a terminal.

The *tcdrain*( ) function may fail if:

[EIO] The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

tcflush( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, **<unistd.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the final paragraph is no longer conditional on _POSIX_JOB_CONTROL. This is a FIPS requirement.

46245 • The [EIO] error is added.

NAME

tcflow — suspend or restart the transmission or reception of data

SYNOPSIS

```
#include <termios.h>

int tcflow(int fildes, int action);
```

DESCRIPTION

The *tcflow*( ) function shall suspend or restart transmission or reception of data on the object referred to by *fildes*, depending on the value of *action*. The *fildes* argument is an open file descriptor associated with a terminal.

- If *action* is TCOOFF, output shall be suspended.
- If *action* is TCOON, suspended output shall be restarted.
- If *action* is TCIOFF, the system shall transmit a STOP character, which is intended to cause the terminal device to stop transmitting data to the system.
- If *action* is TCION, the system shall transmit a START character, which is intended to cause the terminal device to start transmitting data to the system.

The default on the opening of a terminal file is that neither its input nor its output are suspended.

Attempts to use *tcflow*( ) from a process which is a member of a background process group on a *fildes* associated with its controlling terminal, shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process shall be allowed to perform the operation, and no signal is sent.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcflow*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] The *action* argument is not a supported value.

[ENOTTY] The file associated with *fildes* is not a terminal.

The *tcflow*( ) function may fail if:

[EIO] The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

tcsendbreak( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, **<unistd.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] error is added.

46296 **NAME**

46297 tcflush — flush non-transmitted output data, non-read input data, or both

46298 **SYNOPSIS**

46299 `#include <termios.h>`

46300 `int tcflush(int fildes, int queue_selector);`

46301 **DESCRIPTION**

46302 Upon successful completion, *tcflush*( ) shall discard data written to the object referred to by *fildes*
46303 (an open file descriptor associated with a terminal) but not transmitted, or data received but not
46304 read, depending on the value of *queue_selector*:

46305 • If *queue_selector* is TCIFLUSH, it shall flush data received but not read.

46306 • If *queue_selector* is TCOFLUSH, it shall flush data written but not transmitted.

46307 • If *queue_selector* is TCIOFLUSH, it shall flush both data received but not read and data
46308 written but not transmitted.

46309 Attempts to use *tcflush*( ) from a process which is a member of a background process group on a
46310 *fildes* associated with its controlling terminal shall cause the process group to be sent a SIGTTOU
46311 signal. If the calling process is blocking or ignoring SIGTTOU signals, the process shall be
46312 allowed to perform the operation, and no signal is sent.

46313 **RETURN VALUE**

46314 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
46315 indicate the error.

46316 **ERRORS**

46317 The *tcflush*( ) function shall fail if:

46318 [EBADF] The *fildes* argument is not a valid file descriptor.

46319 [EINVAL] The *queue_selector* argument is not a supported value.

46320 [ENOTTY] The file associated with *fildes* is not a terminal.

46321 The *tcflush*( ) function may fail if:

46322 [EIO] The process group of the writing process is orphaned, and the writing process
46323 is not ignoring or blocking SIGTTOU.

46324 **EXAMPLES**

46325 None.

46326 **APPLICATION USAGE**

46327 None.

46328 **RATIONALE**

46329 None.

46330 **FUTURE DIRECTIONS**

46331 None.

46332 **SEE ALSO**

46333 *tcdrain*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, **<unistd.h>**, the
46334 Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

46335 **CHANGE HISTORY**

46336 First released in Issue 3.

46337 Entry included for alignment with the POSIX.1-1988 standard.

46338 **Issue 6**

46339 The Open Group Corrigendum U035/1 is applied. In the ERRORS and APPLICATION USAGE
46340 sections, references to *tcflow*( ) are replaced with *tcflush*( ).

46341 The following new requirements on POSIX implementations derive from alignment with the
46342 Single UNIX Specification:

46343 • In the DESCRIPTION, the final paragraph is no longer conditional on
46344 _POSIX_JOB_CONTROL. This is a FIPS requirement.

46345 • The [EIO] error is added.

46346 **NAME**

46347 tcgetattr — get the parameters associated with the terminal

46348 **SYNOPSIS**

46349 `#include <termios.h>`

46350 `int tcgetattr(int fildes, struct termios *termios_p);`

46351 **DESCRIPTION**

46352 The *tcgetattr*( ) function shall get the parameters associated with the terminal referred to by *fildes*
46353 and store them in the **termios** structure referenced by *termios_p*. The *fildes* argument is an open
46354 file descriptor associated with a terminal.

46355 The *termios_p* argument is a pointer to a **termios** structure.

46356 The *tcgetattr*( ) operation is allowed from any process.

46357 If the terminal device supports different input and output baud rates, the baud rates stored in
46358 the **termios** structure returned by *tcgetattr*( ) shall reflect the actual baud rates, even if they are
46359 equal. If differing baud rates are not supported, the rate returned as the output baud rate shall be
46360 the actual baud rate. If the terminal device does not support split baud rates, the input baud rate
46361 stored in the **termios** structure shall be the output rate (as one of the symbolic values).

46362 **RETURN VALUE**

46363 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
46364 indicate the error.

46365 **ERRORS**

46366 The *tcgetattr*( ) function shall fail if:

46367 [EBADF] The *fildes* argument is not a valid file descriptor.

46368 [ENOTTY] The file associated with *fildes* is not a terminal.

46369 **EXAMPLES**

46370 None.

46371 **APPLICATION USAGE**

46372 None.

46373 **RATIONALE**

46374 Care must be taken when changing the terminal attributes. Applications should always do a
46375 *tcgetattr*( ), save the **termios** structure values returned, and then do a *tcsetattr*( ) changing only
46376 the necessary fields. The application should use the values saved from the *tcgetattr*( ) to reset the
46377 terminal state whenever it is done with the terminal. This is necessary because terminal
46378 attributes apply to the underlying port and not to each individual open instance; that is, all
46379 processes that have used the terminal see the latest attribute changes.

46380 A program that uses these functions should be written to catch all signals and take other
46381 appropriate actions to ensure that when the program terminates, whether planned or not, the
46382 terminal device's state is restored to its original state.

46383 Existing practice dealing with error returns when only part of a request can be honored is based
46384 on calls to the *ioctl*( ) function. In historical BSD and System V implementations, the
46385 corresponding *ioctl*( ) returns zero if the requested actions were semantically correct, even if
46386 some of the requested changes could not be made. Many existing applications assume this
46387 behavior and would no longer work correctly if the return value were changed from zero to –1
46388 in this case.

Note that either specification has a problem. When zero is returned, it implies everything succeeded even if some of the changes were not made. When –1 is returned, it implies everything failed even though some of the changes were made.

Applications that need all of the requested changes made to work properly should follow *tcsetattr*( ) with a call to *tcgetattr*( ) and compare the appropriate field values.

FUTURE DIRECTIONS

None.

SEE ALSO

tcsetattr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

In the DESCRIPTION, the rate returned as the input baud rate shall be the output rate. Previously, the number zero was also allowed but was obsolescent.

NAME

tcgetpgrp — get the foreground process group ID

SYNOPSIS

```
#include <unistd.h>

pid_t tcgetpgrp(int fildes);
```

DESCRIPTION

The *tcgetpgrp*( ) function shall return the value of the process group ID of the foreground process group associated with the terminal.

If there is no foreground process group, *tcgetpgrp*( ) shall return a value greater than 1 that does not match the process group ID of any existing process group.

The *tcgetpgrp*( ) function is allowed from a process that is a member of a background process group; however, the information may be subsequently changed by a process that is a member of a foreground process group.

RETURN VALUE

Upon successful completion, *tcgetpgrp*( ) shall return the value of the process group ID of the foreground process associated with the terminal. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcgetpgrp*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

[ENOTTY] The calling process does not have a controlling terminal, or the file is not the controlling terminal.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

setsid( ), *setpgid*( ), *tcsetpgrp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

- In the DESCRIPTION, text previously conditional on support for _POSIX_JOB_CONTROL is now mandatory. This is a FIPS requirement.

NAME

tcgetsid — get process group ID for session leader for controlling terminal

SYNOPSIS

XSI
```
#include <termios.h>

pid_t tcgetsid(int fildes);
```

DESCRIPTION

The *tcgetsid*( ) function shall obtain the process group ID of the session for which the terminal specified by *fildes* is the controlling terminal.

RETURN VALUE

Upon successful completion, *tcgetsid*( ) shall return the process group ID associated with the terminal. Otherwise, a value of (**pid_t**)–1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcgetsid*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

[ENOTTY] The calling process does not have a controlling terminal, or the file is not the controlling terminal.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The [EACCES] error has been removed from the list of mandatory errors, and the description of [ENOTTY] has been reworded.

46483 **NAME**

46484 tcsendbreak — send a ‘‘break’’ for a specific duration

46485 **SYNOPSIS**

46486
```
#include <termios.h>
```

46487
```
int tcsendbreak(int fildes, int duration);
```

46488 **DESCRIPTION**

46489 If the terminal is using asynchronous serial data transmission, *tcsendbreak*( ) shall cause |
46490 transmission of a continuous stream of zero-valued bits for a specific duration. If *duration* is 0, it
46491 shall cause transmission of zero-valued bits for at least 0,25 seconds, and not more than 0,5
46492 seconds. If *duration* is not 0, it shall send zero-valued bits for an implementation-defined period
46493 of time.

46494 The *fildes* argument is an open file descriptor associated with a terminal. |

46495 If the terminal is not using asynchronous serial data transmission, it is implementation-defined |
46496 whether *tcsendbreak*( ) sends data to generate a break condition or returns without taking any
46497 action.

46498 Attempts to use *tcsendbreak*( ) from a process which is a member of a background process group
46499 on a *fildes* associated with its controlling terminal shall cause the process group to be sent a |
46500 SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process |
46501 shall be allowed to perform the operation, and no signal is sent. |

46502 **RETURN VALUE**

46503 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
46504 indicate the error.

46505 **ERRORS**

46506 The *tcsendbreak*( ) function shall fail if:

46507 [EBADF] The *fildes* argument is not a valid file descriptor.

46508 [ENOTTY] The file associated with *fildes* is not a terminal.

46509 The *tcsendbreak*( ) function may fail if:

46510 [EIO] The process group of the writing process is orphaned, and the writing process
46511 is not ignoring or blocking SIGTTOU.

46512 **EXAMPLES**

46513 None.

46514 **APPLICATION USAGE**

46515 None.

46516 **RATIONALE**

46517 None.

46518 **FUTURE DIRECTIONS**

46519 None.

46520 **SEE ALSO**

46521 The Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, **<unistd.h>**, the Base
46522 Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

46523 **CHANGE HISTORY**

46524 First released in Issue 3.

46525 Entry included for alignment with the POSIX.1-1988 standard.

46526 **Issue 6**

46527 The following new requirements on POSIX implementations derive from alignment with the
46528 Single UNIX Specification:

46529 • In the DESCRIPTION, text previously conditional on _POSIX_JOB_CONTROL is now
46530 mandated. This is a FIPS requirement.

46531 • The [EIO] error is added.

NAME

tcsetattr — set the parameters associated with the terminal

SYNOPSIS

```
#include <termios.h>

int tcsetattr(int fildes, int optional_actions,
    const struct termios *termios_p);
```

DESCRIPTION

The *tcsetattr*( ) function shall set the parameters associated with the terminal referred to by the open file descriptor *fildes* (an open file descriptor associated with a terminal) from the **termios** structure referenced by *termios_p* as follows:

- If *optional_actions* is TCSANOW, the change shall occur immediately.
- If *optional_actions* is TCSADRAIN, the change shall occur after all output written to *fildes* is transmitted. This function should be used when changing parameters that affect output.
- If *optional_actions* is TCSAFLUSH, the change shall occur after all output written to *fildes* is transmitted, and all input so far received but not read shall be discarded before the change is made.

If the output baud rate stored in the **termios** structure pointed to by *termios_p* is the zero baud rate, B0, the modem control lines shall no longer be asserted. Normally, this shall disconnect the line.

If the input baud rate stored in the **termios** structure pointed to by *termios_p* is 0, the input baud rate given to the hardware is the same as the output baud rate stored in the **termios** structure.

The *tcsetattr*( ) function shall return successfully if it was able to perform any of the requested actions, even if some of the requested actions could not be performed. It shall set all the attributes that the implementation supports as requested and leaves all the attributes not supported by the implementation unchanged. If no part of the request can be honored, it shall return −1 and set *errno* to [EINVAL]. If the input and output baud rates differ and are a combination that is not supported, neither baud rate shall be changed. A subsequent call to *tcgetattr*( ) shall return the actual state of the terminal device (reflecting both the changes made and not made in the previous *tcsetattr*( ) call). The *tcsetattr*( ) function shall not change the values found in the **termios** structure under any circumstances.

The effect of *tcsetattr*( ) is undefined if the value of the **termios** structure pointed to by *termios_p* was not derived from the result of a call to *tcgetattr*( ) on *fildes*; an application should modify only fields and flags defined by this volume of IEEE Std 1003.1-200x between the call to *tcgetattr*( ) and *tcsetattr*( ), leaving all other fields and flags unmodified.

No actions defined by this volume of IEEE Std 1003.1-200x, other than a call to *tcsetattr*( ) or a close of the last file descriptor in the system associated with this terminal device, shall cause any of the terminal attributes defined by this volume of IEEE Std 1003.1-200x to change.

If *tcsetattr*( ) is called from a process which is a member of a background process group on a *fildes* associated with its controlling terminal:

- If the calling process is blocking or ignoring SIGTTOU signals, the operation completes normally and no signal is sent.
- Otherwise, a SIGTTOU signal shall be sent to the process group.

46574 **RETURN VALUE**

46575 Upon successful completion, 0 shall be shall be returned. Otherwise, –1 shall be returned and |
46576 *errno* set to indicate the error.

46577 **ERRORS**

46578 The *tcsetattr*( ) function shall fail if:

46579 [EBADF] The *fildes* argument is not a valid file descriptor.

46580 [EINTR] A signal interrupted *tcsetattr*( ).

46581 [EINVAL] The *optional_actions* argument is not a supported value, or an attempt was
46582 made to change an attribute represented in the **termios** structure to an
46583 unsupported value.

46584 [ENOTTY] The file associated with *fildes* is not a terminal.

46585 The *tcsetattr*( ) function may fail if:

46586 [EIO] The process group of the writing process is orphaned, and the writing process
46587 is not ignoring or blocking SIGTTOU.

46588 **EXAMPLES**

46589 None.

46590 **APPLICATION USAGE**

46591 If trying to change baud rates, applications should call *tcsetattr*( ) then call *tcgetattr*( ) in order to
46592 determine what baud rates were actually selected.

46593 **RATIONALE**

46594 The *tcsetattr*( ) function can be interrupted in the following situations:

46595 • It is interrupted while waiting for output to drain.

46596 • It is called from a process in a background process group and SIGTTOU is caught.

46597 See also the RATIONALE section in *tcgetattr*( ).

46598 **FUTURE DIRECTIONS**

46599 Using an input baud rate of 0 to set the input rate equal to the output rate may not necessarily be
46600 supported in a future version of this volume of IEEE Std 1003.1-200x.

46601 **SEE ALSO**

46602 *cfgetispeed*( ), *tcgetattr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**,
46603 **<unistd.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal
46604 Interface

46605 **CHANGE HISTORY**

46606 First released in Issue 3.

46607 Entry included for alignment with the POSIX.1-1988 standard.

46608 **Issue 6**

46609 The following new requirements on POSIX implementations derive from alignment with the
46610 Single UNIX Specification:

46611 • In the DESCRIPTION, text previously conditional on _POSIX_JOB_CONTROL is now
46612 mandated. This is a FIPS requirement.

46613 • The [EIO] error is added.

46614 In the DESCRIPTION, the text describing use of *tcsetattr*( ) from a process which is a member of |
46615 a background process group is clarified.

NAME

tcsetpgrp — set the foreground process group ID

SYNOPSIS

```
#include <unistd.h>

int tcsetpgrp(int fildes, pid_t pgid_id);
```

DESCRIPTION

If the process has a controlling terminal, *tcsetpgrp*( ) shall set the foreground process group ID associated with the terminal to *pgid_id*. The application shall ensure that the file associated with *fildes* is the controlling terminal of the calling process and the controlling terminal is currently associated with the session of the calling process. The application shall ensure that the value of *pgid_id* matches a process group ID of a process in the same session as the calling process.

Attempts to use *tcsetpgrp*( ) from a process which is a member of a background process group on a *fildes* associated with its controlling terminal shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process shall be allowed to perform the operation, and no signal is sent.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcsetpgrp*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] This implementation does not support the value in the *pgid_id* argument.

[ENOTTY] The calling process does not have a controlling terminal, or the file is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

[EPERM] The value of *pgid_id* is a value supported by the implementation, but does not match the process group ID of a process in the same session as the calling process.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

tcgetpgrp( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the DESCRIPTION and ERRORS sections, text previously conditional on _POSIX_JOB_CONTROL is now mandated. This is a FIPS requirement.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The Open Group Corrigendum U047/4 is applied.

NAME

tdelete, tfind, tsearch, twalk — manage a binary search tree

SYNOPSIS

XSI
```
#include <search.h>

void *tdelete(const void *restrict key, void **restrict rootp,
    int(*compar)(const void *, const void *));
void *tfind(const void *key, void *const *rootp,
    int(*compar)(const void *, const void *));
void *tsearch(const void *key, void **rootp,
    int (*compar)(const void *, const void *));
void twalk(const void *root,
    void (*action)(const void *, VISIT, int));
```

DESCRIPTION

The *tdelete*(), *tfind*(), *tsearch*(), and *twalk*() functions manipulate binary search trees. Comparisons are made with a user-supplied routine, the address of which is passed as the *compar* argument. This routine is called with two arguments, the pointers to the elements being compared. The application shall ensure that the user-supplied routine returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to, or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The *tsearch*() function shall build and access the tree. The *key* argument is a pointer to an element to be accessed or stored. If there is a node in the tree whose element is equal to the value pointed to by *key*, a pointer to this found node shall be returned. Otherwise, the value pointed to by *key* shall be inserted (that is, a new node is created and the value of *key* is copied to this node), and a pointer to this node returned. Only pointers are copied, so the application shall ensure that the calling routine stores the data. The *rootp* argument points to a variable that points to the root node of the tree. A null pointer value for the variable pointed to by *rootp* denotes an empty tree; in this case, the variable shall be set to point to the node which shall be at the root of the new tree.

Like *tsearch*(), *tfind*() shall search for a node in the tree, returning a pointer to it if found. However, if it is not found, *tfind*() shall return a null pointer. The arguments for *tfind*() are the same as for *tsearch*().

The *tdelete*() function shall delete a node from a binary search tree. The arguments are the same as for *tsearch*(). The variable pointed to by *rootp* shall be changed if the deleted node was the root of the tree. The *tdelete*() function shall return a pointer to the parent of the deleted node, or a null pointer if the node is not found.

The *twalk*() function shall traverse a binary search tree. The *root* argument is a pointer to the root node of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) The argument *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument shall be the address of the node being visited. The structure pointed to by this argument is unspecified and shall not be modified by the application, but it shall be possible to cast a pointer-to-node into a pointer-to-pointer-to-element to access the element stored in the node. The second argument shall be a value from an enumeration data type:

```
typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

(defined in **<search.h>**), depending on whether this is the first, second, or third time that the node is visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument shall be the level of the node in the tree, with the root being level 0.

If the calling function alters the pointer to the root, the result is undefined.

RETURN VALUE

If the node is found, both *tsearch*( ) and *tfind*( ) shall return a pointer to it. If not, *tfind*( ) shall return a null pointer, and *tsearch*( ) shall return a pointer to the inserted item.

A null pointer shall be returned by *tsearch*( ) if there is not enough space available to create a new node.

A null pointer shall be returned by *tdelete*( ), *tfind*( ), and *tsearch*( ) if *rootp* is a null pointer on entry.

The *tdelete*( ) function shall return a pointer to the parent of the deleted node, or a null pointer if the node is not found.

The *twalk*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

```
#include <search.h>
#include <string.h>
#include <stdio.h>

#define STRSZ    10000
#define NODSZ    500

struct node {      /* Pointers to these are stored in the tree. */
    char    *string;
    int     length;
};

char   string_space[STRSZ];  /* Space to store strings. */
struct node nodes[NODSZ];    /* Nodes to store. */
void  *root = NULL;          /* This points to the root. */

int main(int argc, char *argv[])
{
    char   *strptr = string_space;
    struct node    *nodeptr = nodes;
    void   print_node(const void *, VISIT, int);
    int    i = 0, node_compare(const void *, const void *);

    while (gets(strptr) != NULL && i++ < NODSZ)  {
        /* Set node. */
        nodeptr->string = strptr;
        nodeptr->length = strlen(strptr);
        /* Put node into the tree. */
        (void) tsearch((void *)nodeptr, (void **)&root,
            node_compare);
```

```
46760               /* Adjust pointers, so we do not overwrite tree. */
46761               strptr += nodeptr->length + 1;
46762               nodeptr++;
46763           }
46764           twalk(root, print_node);
46765           return 0;
46766       }
46767       /*
46768        *  This routine compares two nodes, based on an
46769        *  alphabetical ordering of the string field.
46770        */
46771       int
46772       node_compare(const void *node1, const void *node2)
46773       {
46774           return strcmp(((const struct node *) node1)->string,
46775               ((const struct node *) node2)->string);
46776       }
46777       /*
46778        *  This routine prints out a node, the second time
46779        *  twalk encounters it or if it is a leaf.
46780        */
46781       void
46782       print_node(const void *ptr, VISIT order, int level)
46783       {
46784           const struct node *p = *(const struct node **) ptr;
46785           if (order == postorder || order == leaf)  {
46786               (void) printf("string = %s,  length = %d\n",
46787                   p->string, p->length);
46788           }
46789       }
```

46790 **APPLICATION USAGE**

46791 The *root* argument to *twalk*( ) is one level of indirection less than the *rootp* arguments to *tdelete*( )
46792 and *tsearch*( ).

46793 There are two nomenclatures used to refer to the order in which tree nodes are visited. The
46794 *tsearch*( ) function uses **preorder**, **postorder**, and **endorder** to refer respectively to visiting a node
46795 before any of its children, after its left child and before its right, and after both its children. The
46796 alternative nomenclature uses **preorder**, **inorder**, and **postorder** to refer to the same visits, which
46797 could result in some confusion over the meaning of **postorder**.

46798 **RATIONALE**

46799 None.

46800 **FUTURE DIRECTIONS**

46801 None.

46802 **SEE ALSO**

46803 *hcreate*( ), *lsearch*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<search.h>**

46804 **CHANGE HISTORY**

46805 First released in Issue 1. Derived from Issue 1 of the SVID.

46806 **Issue 5**

46807 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
46808 previous issues.

46809 **Issue 6**

46810 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

46811 The **restrict** keyword is added to the *tdelete*() prototype for alignment with the
46812 ISO / IEC 9899: 1999 standard.

NAME

telldir — current location of a named directory stream

SYNOPSIS

XSI

```
#include <dirent.h>

long telldir(DIR *dirp);
```

DESCRIPTION

The *telldir*( ) function shall obtain the current location associated with the directory stream specified by *dirp*.

If the most recent operation on the directory stream was a *seekdir*( ), the directory position returned from the *telldir*( ) shall be the same as that supplied as a *loc* argument for *seekdir*( ).

RETURN VALUE

Upon successful completion, *telldir*( ) shall return the current location of the specified directory stream.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

opendir( ), *readdir*( ), *seekdir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**

CHANGE HISTORY

First released in Issue 2.

NAME

tempnam — create a name for a temporary file

SYNOPSIS

XSI
```
#include <stdio.h>

char *tempnam(const char *dir, const char *pfx);
```

DESCRIPTION

The *tempnam*( ) function shall generate a pathname that may be used for a temporary file.

The *tempnam*( ) function allows the user to control the choice of a directory. The *dir* argument points to the name of the directory in which the file is to be created. If *dir* is a null pointer or points to a string which is not a name for an appropriate directory, the path prefix defined as P_tmpdir in the **<stdio.h>** header shall be used. If that directory is not accessible, an implementation-defined directory may be used.

Many applications prefer their temporary files to have certain initial letter sequences in their names. The *pfx* argument should be used for this. This argument may be a null pointer or point to a string of up to five bytes to be used as the beginning of the filename.

Some implementations of *tempnam*( ) may use *tmpnam*( ) internally. On such implementations, if called more than {TMP_MAX} times in a single process, the behavior is implementation-defined.

RETURN VALUE

Upon successful completion, *tempnam*( ) shall allocate space for a string, put the generated pathname in that space, and return a pointer to it. The pointer shall be suitable for use in a subsequent call to *free*( ). Otherwise, it shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *tempnam*( ) function shall fail if:

[ENOMEM] Insufficient storage space is available.

EXAMPLES

Generating a Pathname

The following example generates a pathname for a temporary file in directory **/tmp**, with the prefix *file*. After the filename has been created, the call to *free*( ) deallocates the space used to store the filename.

```
#include <stdio.h>
#include <stdlib.h>
...
char *directory = "/tmp";
char *fileprefix = "file";
char *file;

file = tempnam(directory, fileprefix);
free(file);
```

APPLICATION USAGE

This function only creates pathnames. It is the application's responsibility to create and remove the files. Between the time a pathname is created and the file is opened, it is possible for some other process to create a file with the same name. Applications may find *tmpfile*( ) more useful.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), *free*( ), *open*( ), *tmpfile*( ), *tmpnam*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

46896 **NAME**

46897 tfind — search binary search tree

46898 **SYNOPSIS**

46899 XSI
```
#include <search.h>

void *tfind(const void *key, void *const *rootp,
    int (*compar)(const void *, const void *));
```
46900
46901
46902

46903 **DESCRIPTION**

46904 Refer to *tdelete*( ).

NAME

tgamma, tgammaf, tgammal — compute gamma( ) function

SYNOPSIS

```
#include <math.h>

double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the *gamma*( ) function of *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return *Gamma*(*x*).

If *x* is a negative integer, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If the correct value would cause overflow, a range error shall occur and *tgamma*( ), *tgammaf*( ), and *tgammal*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, or HUGE_VALL, respectively.

MX If *x* is NaN, a NaN shall be returned.

If *x* is +Inf, *x* shall be returned.

If *x* is ±0, a pole error shall occur, and *tgamma*( ), *tgammaf*( ), and *tgammal*( ) shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL, respectively.

If *x* is –Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS

These functions shall fail if:

MX Domain Error — The value of *x* is a negative integer, or *x* is –Inf.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERRNOEXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

MX Pole Error — The value of *x* is zero.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised.

Range Error — The value overflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

For IEEE Std 754-1985 **double**, overflow happens when $0 < x < 1/\text{DBL\_MAX}$, and $171.7 < x$. Overflow also happens near negative integers.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

This function is named *tgamma*( ) in order to avoid conflicts with the historical *gamma*( ) and *lgamma*( ) functions.

FUTURE DIRECTIONS

It is possible that the error response for a negative integer argument may be changed to a pole error and a return value of ±Inf.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *lgamma*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

time — get time

SYNOPSIS

```
#include <time.h>

time_t time(time_t *tloc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *time*( ) function shall return the value of time in seconds since the Epoch.

The *tloc* argument points to an area where the return value is also stored. If *tloc* is a null pointer, no value is stored.

RETURN VALUE

Upon successful completion, *time*( ) shall return the value of time. Otherwise, (**time_t**)–1 shall be returned.

ERRORS

No errors are defined.

EXAMPLES

Getting the Current Time

The following example uses the *time*( ) function to calculate the time elapsed, in seconds, since January 1, 1970 0:00 UTC, *localtime*( ) to convert that value to a broken-down time, and *asctime*( ) to convert the broken-down time values into a printable string.

```
#include <stdio.h>
#include <time.h>

main()
{
time_t result;

    result = time(NULL);
    printf("%s%ld secs since the Epoch\n",
        asctime(localtime(&result)),
            (long)result);
    return(0);
}
```

This example writes the current time to *stdout* in a form like this:

```
Wed Jun 26 10:32:15 1996
835810335 secs since the Epoch
```

47005 **Timing an Event**

47006 The following example gets the current time, prints it out in the user's format, and prints the
47007 number of minutes to an event being timed.

```
47008 #include <time.h>
47009 #include <stdio.h>
47010 ...
47011 time_t now;
47012 int minutes_to_event;
47013 ...
47014 time(&now);
47015 minutes_to_event = ...;
47016 printf("The time is ");
47017 puts(asctime(localtime(&now)));
47018 printf("There are %d minutes to the event.\n",
47019     minutes_to_event);
47020 ...
```

47021 **APPLICATION USAGE**

47022 None.

47023 **RATIONALE**

47024 The *time*( ) function returns a value in seconds (type **time_t**) while *times*( ) returns a set of values
47025 in clock ticks (type **clock_t**). Some historical implementations, such as 4.3 BSD, have
47026 mechanisms capable of returning more precise times (see below). A generalized timing scheme
47027 to unify these various timing mechanisms has been proposed but not adopted.

47028 Implementations in which **time_t** is a 32-bit signed integer (many historical implementations)
47029 fail in the year 2038. IEEE Std 1003.1-200x does not address this problem. However, the use of
47030 the **time_t** type is mandated in order to ease the eventual fix.

47031 The use of the **<time.h>**, header instead of **<sys/types.h>**, allows compatibility with the ISO C
47032 standard.

47033 Many historical implementations (including Version 7) and the 1984 /usr/group standard use
47034 **long** instead of **time_t**. This volume of IEEE Std 1003.1-200x uses the latter type in order to agree
47035 with the ISO C standard.

47036 4.3 BSD includes *time*( ) only as an alternate function to the more flexible *gettimeofday*( ) function.

47037 **FUTURE DIRECTIONS**

47038 In a future version of this volume of IEEE Std 1003.1-200x, **time_t** is likely to be required to be
47039 capable of representing times far in the future. Whether this will be mandated as a 64-bit type or
47040 a requirement that a specific date in the future be representable (for example, 10000 AD) is not
47041 yet determined. Systems purchased after the approval of this volume of IEEE Std 1003.1-200x
47042 should be evaluated to determine whether their lifetime will extend past 2038.

47043 **SEE ALSO**

47044 *asctime*( ), *clock*( ), *ctime*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *utime*( ),
47045 the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

47046 **CHANGE HISTORY**

47047 First released in Issue 1. Derived from Issue 1 of the SVID.

47048 **Issue 6**

47049 Extensions beyond the ISO C standard are now marked. |

47050 The EXAMPLES, RATIONALE, and FUTURE DIRECTIONS sections are added. |

47051 **NAME**

47052 timer_create — create a per-process timer (**REALTIME**)

47053 **SYNOPSIS**

47054 TMR
```
#include <signal.h>
#include <time.h>

int timer_create(clockid_t clockid, struct sigevent *restrict evp,
    timer_t *restrict timerid);
```
47058

47059 **DESCRIPTION**

47060 The *timer_create*( ) function shall create a per-process timer using the specified clock, *clock_id*, as
47061 the timing base. The *timer_create*( ) function shall return, in the location referenced by *timerid*, a
47062 timer ID of type **timer_t** used to identify the timer in timer requests. This timer ID shall be
47063 unique within the calling process until the timer is deleted. The particular clock, *clock_id*, is
47064 defined in **<time.h>**. The timer whose ID is returned shall be in a disarmed state upon return
47065 from *timer_create*( ).

47066 The *evp* argument, if non-NULL, points to a **sigevent** structure. This structure, allocated by the
47067 application, defines the asynchronous notification to occur as specified in Section 2.4.1 (on page
47068 478) when the timer expires. If the *evp* argument is NULL, the effect is as if the *evp* argument
47069 pointed to a **sigevent** structure with the *sigev_notify* member having the value SIGEV_SIGNAL,
47070 the *sigev_signo* having a default signal number, and the *sigev_value* member having the value of
47071 the timer ID.

47072 Each implementation shall define a set of clocks that can be used as timing bases for per-process
47073 MON timers. All implementations shall support a *clock_id* of CLOCK_REALTIME. If the Monotonic
47074 Clock option is supported, implementations shall support a *clock_id* of CLOCK_MONOTONIC.

47075 Per-process timers shall not be inherited by a child process across a *fork*( ) and shall be disarmed
47076 and deleted by an *exec*.

47077 CPT If _POSIX_CPUTIME is defined, implementations shall support *clock_id* values representing the
47078 CPU-time clock of the calling process.

47079 TCT If _POSIX_THREAD_CPUTIME is defined, implementations shall support *clock_id* values
47080 representing the CPU-time clock of the calling thread.

47081 CPT|TCT It is implementation-defined whether a *timer_create*( ) function will succeed if the value defined
47082 by *clock_id* corresponds to the CPU-time clock of a process or thread different from the process
47083 or thread invoking the function.

47084 **RETURN VALUE**

47085 If the call succeeds, *timer_create*( ) shall return zero and update the location referenced by *timerid*
47086 to a **timer_t**, which can be passed to the per-process timer calls. If an error occurs, the function
47087 shall return a value of –1 and set *errno* to indicate the error. The value of *timerid* is undefined if
47088 an error occurs.

47089 **ERRORS**

47090 The *timer_create*( ) function shall fail if:

47091 [EAGAIN] The system lacks sufficient signal queuing resources to honor the request.

47092 [EAGAIN] The calling process has already created all of the timers it is allowed by this
47093 implementation.

47094 [EINVAL] The specified clock ID is not defined.

47095 CPT|TCT [ENOTSUP] The implementation does not support the creation of a timer attached to the
47096 CPU-time clock that is specified by *clock_id* and associated with a process or
47097 thread different from the process or thread invoking *timer_create*( ).

47098 **EXAMPLES**

47099 None.

47100 **APPLICATION USAGE**

47101 None.

47102 **RATIONALE**

47103 **Periodic Timer Overrun and Resource Allocation**

47104 The specified timer facilities may deliver realtime signals (that is, queued signals) on |
47105 implementations that support this option. Since realtime applications cannot afford to lose |
47106 notifications of asynchronous events, like timer expirations or asynchronous I/O completions, it
47107 must be possible to ensure that sufficient resources exist to deliver the signal when the event
47108 occurs. In general, this is not a difficulty because there is a one-to-one correspondence between a
47109 request and a subsequent signal generation. If the request cannot allocate the signal delivery
47110 resources, it can fail the call with an [EAGAIN] error.

47111 Periodic timers are a special case. A single request can generate an unspecified number of |
47112 signals. This is not a problem if the requesting process can service the signals as fast as they are |
47113 generated, thus making the signal delivery resources available for delivery of subsequent
47114 periodic timer expiration signals. But, in general, this cannot be assured—processing of periodic
47115 timer signals may ‘‘overrun’’; that is, subsequent periodic timer expirations may occur before the
47116 currently pending signal has been delivered.

47117 Also, for signals, according to the POSIX.1-1990 standard, if subsequent occurrences of a
47118 pending signal are generated, it is implementation-defined whether a signal is delivered for each
47119 occurrence. This is not adequate for some realtime applications. So a mechanism is required to
47120 allow applications to detect how many timer expirations were delayed without requiring an
47121 indefinite amount of system resources to store the delayed expirations.

47122 The specified facilities provide for an overrun count. The overrun count is defined as the number
47123 of extra timer expirations that occurred between the time a timer expiration signal is generated
47124 and the time the signal is delivered. The signal-catching function, if it is concerned with
47125 overruns, can retrieve this count on entry. With this method, a periodic timer only needs one
47126 ‘‘signal queuing resource’’ that can be allocated at the time of the *timer_create*( ) function call.

47127 A function is defined to retrieve the overrun count so that an application need not allocate static
47128 storage to contain the count, and an implementation need not update this storage
47129 asynchronously on timer expirations. But, for some high-frequency periodic applications, the
47130 overhead of an additional system call on each timer expiration may be prohibitive. The
47131 functions, as defined, permit an implementation to maintain the overrun count in user space,
47132 associated with the *timerid*. The *timer_getoverrun*( ) function can then be implemented as a macro
47133 that uses the *timerid* argument (which may just be a pointer to a user space structure containing
47134 the counter) to locate the overrun count with no system call overhead. Other implementations,
47135 less concerned with this class of applications, can avoid the asynchronous update of user space
47136 by maintaining the count in a system structure at the cost of the extra system call to obtain it.

47137 **Timer Expiration Signal Parameters**

47138 The Realtime Signals Extension option supports an application-specific datum that is delivered
47139 to the extended signal handler. This value is explicitly specified by the application, along with
47140 the signal number to be delivered, in a **sigevent** structure. The type of the application-defined
47141 value can be either an integer constant or a pointer. This explicit specification of the value, as
47142 opposed to always sending the timer ID, was selected based on existing practice.

47143 It is common practice for realtime applications (on non-POSIX systems or realtime extended
47144 POSIX systems) to use the parameters of event handlers as the case label of a switch statement
47145 or as a pointer to an application-defined data structure. Since *timer_id*s are dynamically allocated |
47146 by the *timer_create*( ) function, they can be used for neither of these functions without additional
47147 application overhead in the signal handler; for example, to search an array of saved timer IDs to
47148 associate the ID with a constant or application data structure.

47149 **FUTURE DIRECTIONS**

47150 None.

47151 **SEE ALSO**

47152 *clock_getres*( ), *timer_delete*( ), *timer_getoverrun*( ), the Base Definitions volume of
47153 IEEE Std 1003.1-200x, **<time.h>**

47154 **CHANGE HISTORY**

47155 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

47156 **Issue 6**

47157 The *timer_create*( ) function is marked as part of the Timers option.

47158 The [ENOSYS] error condition has been removed as stubs need not be provided if an
47159 implementation does not support the Timers option.

47160 CPU-time clocks are added for alignment with IEEE Std 1003.1d-1999.

47161 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding the
47162 requirement for the CLOCK_MONOTONIC clock under the Monotonic Clock option.

47163 The **restrict** keyword is added to the *timer_create*( ) prototype for alignment with the
47164 ISO / IEC 9899: 1999 standard.

47165 **NAME**

47166 timer_delete — delete a per-process timer (**REALTIME**)

47167 **SYNOPSIS**

47168 TMR
```
#include <time.h>
```

47169
```
int timer_delete(timer_t timerid);
```

47170

47171 **DESCRIPTION**

47172 The *timer_delete*( ) function deletes the specified timer, *timerid*, previously created by the
47173 *timer_create*( ) function. If the timer is armed when *timer_delete*( ) is called, the behavior shall be
47174 as if the timer is automatically disarmed before removal. The disposition of pending signals for
47175 the deleted timer is unspecified.

47176 **RETURN VALUE**

47177 If successful, the *timer_delete*( ) function shall return a value of zero. Otherwise, the function shall
47178 return a value of –1 and set *errno* to indicate the error.

47179 **ERRORS**

47180 The *timer_delete*( ) function shall fail if:

47181 [EINVAL] The timer ID specified by *timerid* is not a valid timer ID.

47182 **EXAMPLES**

47183 None.

47184 **APPLICATION USAGE**

47185 None.

47186 **RATIONALE**

47187 None.

47188 **FUTURE DIRECTIONS**

47189 None.

47190 **SEE ALSO**

47191 *timer_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

47192 **CHANGE HISTORY**

47193 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

47194 **Issue 6**

47195 The *timer_delete*( ) function is marked as part of the Timers option.

47196 The [ENOSYS] error condition has been removed as stubs need not be provided if an
47197 implementation does not support the Timers option.

NAME

timer_getoverrun, timer_gettime, timer_settime — per-process timers (**REALTIME**)

SYNOPSIS

TMR
```
#include <time.h>

int timer_getoverrun(timer_t timerid);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_settime(timer_t timerid, int flags,
    const struct itimerspec *restrict value,
    struct itimerspec *restrict ovalue);
```

DESCRIPTION

The *timer_gettime*( ) function shall store the amount of time until the specified timer, *timerid*, expires and the reload value of the timer into the space pointed to by the *value* argument. The *it_value* member of this structure shall contain the amount of time before the timer expires, or zero if the timer is disarmed. This value is returned as the interval until timer expiration, even if the timer was armed with absolute time. The *it_interval* member of *value* shall contain the reload value last set by *timer_settime*( ).

The *timer_settime*( ) function shall set the time until the next expiration of the timer specified by *timerid* from the *it_value* member of the *value* argument and arms the timer if the *it_value* member of *value* is non-zero. If the specified timer was already armed when *timer_settime*( ) is called, this call shall reset the time until next expiration to the *value* specified. If the *it_value* member of *value* is zero, the timer shall be disarmed. The effect of disarming or resetting a timer with pending expiration notifications is unspecified.

If the flag TIMER_ABSTIME is not set in the argument *flags*, *timer_settime*( ) shall behave as if the time until next expiration is set to be equal to the interval specified by the *it_value* member of *value*. That is, the timer shall expire in *it_value* nanoseconds from when the call is made. If the flag TIMER_ABSTIME is set in the argument *flags*, *timer_settime*( ) shall behave as if the time until next expiration is set to be equal to the difference between the absolute time specified by the *it_value* member of *value* and the current value of the clock associated with *timerid*. That is, the timer shall expire when the clock reaches the value specified by the *it_value* member of *value*. If the specified time has already passed, the function shall succeed and the expiration notification shall be made.

The reload value of the timer shall be set to the value specified by the *it_interval* member of *value*. When a timer is armed with a non-zero *it_interval*, a periodic (or repetitive) timer is specified.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer shall be rounded up to the larger multiple of the resolution. Quantization error shall not cause the timer to expire earlier than the rounded time value.

If the argument *ovalue* is not NULL, the function *timer_settime*( ) shall store, in the location referenced by *ovalue*, a value representing the previous amount of time before the timer would have expired, or zero if the timer was disarmed, together with the previous timer reload value. Timers shall not expire before their scheduled time.

Only a single signal shall be queued to the process for a given timer at any point in time. When a timer for which a signal is still pending expires, no signal shall be queued, and a timer overrun shall occur. When a timer expiration signal is delivered to or accepted by a process, if the RTS implementation supports the Realtime Signals Extension, the *timer_getoverrun*( ) function shall return the timer expiration overrun count for the specified timer. The overrun count returned contains the number of extra timer expirations that occurred between the time the signal was

generated (queued) and when it was delivered or accepted, up to but not including an implementation-defined maximum of {DELAYTIMER_MAX}. If the number of such extra expirations is greater than or equal to {DELAYTIMER_MAX}, then the overrun count shall be set to {DELAYTIMER_MAX}. The value returned by *timer_getoverrun*( ) shall apply to the most recent expiration signal delivery or acceptance for the timer. If no expiration signal has been delivered for the timer, or if the Realtime Signals Extension is not supported, the return value of *timer_getoverrun*( ) is unspecified.

RETURN VALUE

If the *timer_getoverrun*( ) function succeeds, it shall return the timer expiration overrun count as explained above.

If the *timer_gettime*( ) or *timer_settime*( ) functions succeed, a value of 0 shall be returned.

If an error occurs for any of these functions, the value –1 shall be returned, and *errno* set to indicate the error.

ERRORS

The *timer_getoverrun*( ), *timer_gettime*( ), and *timer_settime*( ) functions shall if:

[EINVAL] The *timerid* argument does not correspond to an ID returned by *timer_create*( ) but not yet deleted by *timer_delete*( ).

The *timer_settime*( ) function shall fail if:

[EINVAL] A *value* structure specified a nanosecond value less than zero or greater than or equal to 1,000 million, and the *it_value* member of that structure did not specify zero seconds and nanoseconds.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Practical clocks tick at a finite rate, with rates of 100 Hertz and 1,000 Hertz being common. The inverse of this tick rate is the clock resolution, also called the clock granularity, which in either case is expressed as a time duration, being 10 milliseconds and 1 millisecond respectively for these common rates. The granularity of practical clocks implies that if one reads a given clock twice in rapid succession, one may get the same time value twice; and that timers must wait for the next clock tick after the theoretical expiration time, to ensure that a timer never returns too soon. Note also that the granularity of the clock may be significantly coarser than the resolution of the data format used to set and get time and interval values. Also note that some implementations may choose to adjust time and/or interval values to exactly match the ticks of the underlying clock.

This volume of IEEE Std 1003.1-200x defines functions that allow an application to determine the implementation-supported resolution for the clocks and requires an implementation to document the resolution supported for timers and *nanosleep*( ) if they differ from the supported clock resolution. This is more of a procurement issue than a runtime application issue.

FUTURE DIRECTIONS

None.

47288 **SEE ALSO**

47289 *clock_getres*( ), *timer_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

47290 **CHANGE HISTORY**

47291 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

47292 **Issue 6**

47293 The *timer_getoverrun*( ), *timer_gettime*( ), and *timer_settime*( ) functions are marked as part of the
47294 Timers option.

47295 The [ENOSYS] error condition has been removed as stubs need not be provided if an
47296 implementation does not support the Timers option.

47297 The [EINVAL] error condition is updated to include the following: ‘‘and the *it_value* member of
47298 that structure did not specify zero seconds and nanoseconds.’’ This change is for IEEE PASC
47299 Interpretation 1003.1 #89.

47300 The DESCRIPTION for *timer_getoverrun*( ) is updated to clarify that ‘‘If no expiration signal has
47301 been delivered for the timer, or if the Realtime Signals Extension is not supported, the return
47302 value of *timer_getoverrun*( ) is unspecified’’.

47303 The **restrict** keyword is added to the *timer_settime*( ) prototype for alignment with the
47304 ISO / IEC 9899: 1999 standard.

47305 **NAME**

47306 times — get process and waited-for child process times

47307 **SYNOPSIS**

47308 `#include <sys/times.h>`

47309 `clock_t times(struct tms *buffer);`

47310 **DESCRIPTION**

47311 The *times*( ) function shall fill the **tms** structure pointed to by *buffer* with time-accounting
47312 information. The **tms** structure is defined in **<sys/times.h>**.

47313 All times are measured in terms of the number of clock ticks used.

47314 The times of a terminated child process shall be included in the *tms_cutime* and *tms_cstime*
47315 elements of the parent when *wait*( ) or *waitpid*( ) returns the process ID of this terminated child. If
47316 a child process has not waited for its children, their times shall not be included in its times.

47317 • The *tms_utime* structure member is the CPU time charged for the execution of user
47318 instructions of the calling process.

47319 • The *tms_stime* structure member is the CPU time charged for execution by the system on
47320 behalf of the calling process.

47321 • The *tms_cutime* structure member is the sum of the *tms_utime* and *tms_cutime* times of the
47322 child processes.

47323 • The *tms_cstime* structure member is the sum of the *tms_stime* and *tms_cstime* times of the child
47324 processes.

47325 **RETURN VALUE**

47326 Upon successful completion, *times*( ) shall return the elapsed real time, in clock ticks, since an
47327 arbitrary point in the past (for example, system start-up time). This point does not change from
47328 one invocation of *times*( ) within the process to another. The return value may overflow the
47329 possible range of type **clock_t**. If *times*( ) fails, (**clock_t**)–1 shall be returned and *errno* set to
47330 indicate the error.

47331 **ERRORS**

47332 No errors are defined.

47333 **EXAMPLES**

47334 **Timing a Database Lookup**

47335 The following example defines two functions, *start_clock*( ) and *end_clock*( ), that are used to time
47336 a lookup. It also defines variables of type **clock_t** and **tms** to measure the duration of
47337 transactions. The *start_clock*( ) function saves the beginning times given by the *times*( ) function.
47338 The *end_clock*( ) function gets the ending times and prints the difference between the two times.

```
47339 #include <sys/times.h>
47340 #include <stdio.h>
47341 ...
47342 void start_clock(void);
47343 void end_clock(char *msg);
47344 ...
47345 static clock_t st_time;
47346 static clock_t en_time;
47347 static struct tms st_cpu;
47348 static struct tms en_cpu;
```

```
47349       ...
47350       void
47351       start_clock()
47352       {
47353           st_time = times(&st_cpu);
47354       }

47355       void
47356       end_clock(char *msg)
47357       {
47358           en_time = times(&en_cpu);

47359           printf(msg);
47360           printf("Real Time: %ld, User Time %ld, System Time %ld\n",
47361               en_time - st_time,
47362               en_cpu.tms_utime - st_cpu.tms_utime,
47363               en_cpu.tms_stime - st_cpu.tms_stime);
47364       }
```

47365 **APPLICATION USAGE**

47366 Applications should use *sysconf*(_SC_CLK_TCK) to determine the number of clock ticks per
47367 second as it may vary from system to system.

47368 **RATIONALE**

47369 The accuracy of the times reported is intentionally left unspecified to allow implementations
47370 flexibility in design, from uniprocessor to multi-processor networks.

47371 The inclusion of times of child processes is recursive, so that a parent process may collect the
47372 total times of all of its descendants. But the times of a child are only added to those of its parent
47373 when its parent successfully waits on the child. Thus, it is not guaranteed that a parent process
47374 can always see the total times of all its descendants; see also the discussion of the term *realtime* in
47375 *alarm*( ).

47376 If the type **clock_t** is defined to be a signed 32-bit integer, it overflows in somewhat more than a
47377 year if there are 60 clock ticks per second, or less than a year if there are 100. There are individual
47378 systems that run continuously for longer than that. This volume of IEEE Std 1003.1-200x permits
47379 an implementation to make the reference point for the returned value be the start-up time of the
47380 process, rather than system start-up time.

47381 The term *charge* in this context has nothing to do with billing for services. The operating system
47382 accounts for time used in this way. That information must be correct, regardless of how that
47383 information is used.

47384 **FUTURE DIRECTIONS**

47385 None.

47386 **SEE ALSO**

47387 *exec*, *fork*( ), *sysconf*( ), *time*( ), *wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
47388 **<sys/times.h>**

47389 **CHANGE HISTORY**

47390 First released in Issue 1. Derived from Issue 1 of the SVID.

47391 **NAME**

47392 timezone — difference from UTC and local standard time

47393 **SYNOPSIS**

47394 XSI
```
#include <time.h>
```

47395
```
extern long timezone;
```

47396

47397 **DESCRIPTION**

47398 Refer to *tzset*( ).

47399 **NAME**

47400 tmpfile — create a temporary file

47401 **SYNOPSIS**

47402
```
#include <stdio.h>
```

47403
```
FILE *tmpfile(void);
```

47404 **DESCRIPTION**

47405 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47406 conflict between the requirements described here and the ISO C standard is unintentional. This
47407 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47408 The *tmpfile*( ) function shall create a temporary file and open a corresponding stream. The file |
47409 shall be automatically deleted when all references to the file are closed. The file is opened as in |
47410 *fopen*( ) for update (*w+*).

47411 CX In some implementations, a permanent file may be left behind if the process calling *tmpfile*( ) is
47412 killed while it is processing a call to *tmpfile*( ).

47413 An error message may be written to standard error if the stream cannot be opened.

47414 **RETURN VALUE**

47415 Upon successful completion, *tmpfile*( ) shall return a pointer to the stream of the file that is
47416 CX created. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

47417 **ERRORS**

47418 The *tmpfile*( ) function shall fail if:

47419 CX [EINTR] A signal was caught during *tmpfile*( ).

47420 CX [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

47421 CX [ENFILE] The maximum allowable number of files is currently open in the system.

47422 CX [ENOSPC] The directory or file system which would contain the new file cannot be
47423 expanded.

47424 CX [EOVERFLOW] The file is a regular file and the size of the file cannot be represented correctly
47425 in an object of type **off_t**.

47426 The *tmpfile*( ) function may fail if:

47427 CX [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

47428 CX [ENOMEM] Insufficient storage space is available.

47429 **EXAMPLES**

47430 **Creating a Temporary File**

47431 The following example creates a temporary file for update, and returns a pointer to a stream for
47432 the created file in the *fp* variable.

47433
```
#include <stdio.h>
```
47434
```
...
```
47435
```
FILE *fp;
```
47436
```
fp = tmpfile ();
```

APPLICATION USAGE

It should be possible to open at least {TMP_MAX} temporary files during the lifetime of the program (this limit may be shared with *tmpnam*( )) and there should be no limit on the number simultaneously open other than this limit and any limit on the number of open files ({FOPEN_MAX}).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), *tmpnam*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes in previous issues.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support large files.
- The [EMFILE] optional error condition is added.

The APPLICATION USAGE section is added for alignment with the ISO/IEC 9899:1999 standard.

NAME

tmpnam — create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam(char *s);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *tmpnam*( ) function shall generate a string that is a valid filename and that is not the same as the name of an existing file. The function is potentially capable of generating {TMP_MAX} different strings, but any or all of them may already be in use by existing files and thus not be suitable return values.

The *tmpnam*( ) function generates a different string each time it is called from the same process, up to {TMP_MAX} times. If it is called more than {TMP_MAX} times, the behavior is implementation-defined.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *tmpnam*( ).

CX If the application uses any of the functions guaranteed to be available if either _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS is defined, the application shall ensure that the *tmpnam*( ) function is called with a non-NULL parameter.

RETURN VALUE

Upon successful completion, *tmpnam*( ) shall return a pointer to a string. If no suitable string can be generated, the *tmpnam*( ) function shall return a null pointer.

If the argument *s* is a null pointer, *tmpnam*( ) shall leave its result in an internal static object and return a pointer to that object. Subsequent calls to *tmpnam*( ) may modify the same object. If the argument *s* is not a null pointer, it is presumed to point to an array of at least L_tmpnam **char**s; *tmpnam*( ) shall write its result in that array and shall return the argument as its value.

ERRORS

No errors are defined.

EXAMPLES

Generating a Filename

The following example generates a unique filename and stores it in the array pointed to by *ptr*.

```
#include <stdio.h>
...
char filename[L_tmpnam+1];
char *ptr;

ptr = tmpnam(filename);
```

APPLICATION USAGE

This function only creates filenames. It is the application's responsibility to create and remove the files.

Between the time a pathname is created and the file is opened, it is possible for some other process to create a file with the same name. Applications may find *tmpfile*( ) more useful. |

47506 **RATIONALE**

47507 None.

47508 **FUTURE DIRECTIONS**

47509 None.

47510 **SEE ALSO**

47511 *fopen*( ), *open*( ), *tempnam*( ), *tmpfile*( ), *unlink*( ), the Base Definitions volume of
47512 IEEE Std 1003.1-200x, **<stdio.h>**

47513 **CHANGE HISTORY**

47514 First released in Issue 1. Derived from Issue 1 of the SVID.

47515 **Issue 5**

47516 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

47517 **Issue 6**

47518 Extensions beyond the ISO C standard are now marked.

47519 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

47520 The DESCRIPTION is expanded for alignment with the ISO/IEC 9899: 1999 standard.

47521 **NAME**

47522 toascii — translate integer to a 7-bit ASCII character

47523 **SYNOPSIS**

47524 XSI
```
#include <ctype.h>
```
47525
```
int toascii(int c);
```
47526

47527 **DESCRIPTION**

47528 The *toascii*( ) function shall convert its argument into a 7-bit ASCII character.

47529 **RETURN VALUE**

47530 The *toascii*( ) function shall return the value (*c* &0x7f).

47531 **ERRORS**

47532 No errors are returned.

47533 **EXAMPLES**

47534 None.

47535 **APPLICATION USAGE**

47536 None.

47537 **RATIONALE**

47538 None.

47539 **FUTURE DIRECTIONS**

47540 None.

47541 **SEE ALSO**

47542 *isascii*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**

47543 **CHANGE HISTORY**

47544 First released in Issue 1. Derived from Issue 1 of the SVID.

47545 **NAME**

47546 tolower — transliterate uppercase characters to lowercase

47547 **SYNOPSIS**

47548
```
#include <ctype.h>
```

47549
```
int tolower(int c);
```

47550 **DESCRIPTION**

47551 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47552 conflict between the requirements described here and the ISO C standard is unintentional. This
47553 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47554 The *tolower*( ) function has as a domain a type **int**, the value of which is representable as an
47555 **unsigned char** or the value of EOF. If the argument has any other value, the behavior is
47556 undefined. If the argument of *tolower*( ) represents an uppercase letter, and there exists a
47557 CX corresponding lowercase letter (as defined by character type information in the program locale
47558 category *LC_CTYPE*), the result shall be the corresponding lowercase letter. All other arguments |
47559 in the domain are returned unchanged. |

47560 **RETURN VALUE**

47561 Upon successful completion, *tolower*( ) shall return the lowercase letter corresponding to the
47562 argument passed; otherwise, it shall return the argument unchanged.

47563 **ERRORS**

47564 No errors are defined.

47565 **EXAMPLES**

47566 None.

47567 **APPLICATION USAGE**

47568 None.

47569 **RATIONALE**

47570 None.

47571 **FUTURE DIRECTIONS**

47572 None.

47573 **SEE ALSO**

47574 *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions
47575 volume of IEEE Std 1003.1-200x, Chapter 7, Locale

47576 **CHANGE HISTORY**

47577 First released in Issue 1. Derived from Issue 1 of the SVID.

47578 **Issue 6**

47579 Extensions beyond the ISO C standard are now marked.

47580 **NAME**

47581 toupper — transliterate lowercase characters to uppercase

47582 **SYNOPSIS**

47583
```
#include <ctype.h>
```

47584
```
int toupper(int c);
```

47585 **DESCRIPTION**

47586 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47587 conflict between the requirements described here and the ISO C standard is unintentional. This
47588 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47589 The *toupper*( ) function has as a domain a type **int**, the value of which is representable as an
47590 **unsigned char** or the value of EOF. If the argument has any other value, the behavior is
47591 undefined. If the argument of *toupper*( ) represents a lowercase letter, and there exists a
47592 CX corresponding uppercase letter (as defined by character type information in the program locale
47593 category *LC_CTYPE*), the result shall be the corresponding uppercase letter. All other arguments |
47594 in the domain are returned unchanged. |

47595 **RETURN VALUE**

47596 Upon successful completion, *toupper*( ) shall return the uppercase letter corresponding to the
47597 argument passed.

47598 **ERRORS**

47599 No errors are defined.

47600 **EXAMPLES**

47601 None.

47602 **APPLICATION USAGE**

47603 None.

47604 **RATIONALE**

47605 None.

47606 **FUTURE DIRECTIONS**

47607 None.

47608 **SEE ALSO**

47609 *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions
47610 volume of IEEE Std 1003.1-200x, Chapter 7, Locale

47611 **CHANGE HISTORY**

47612 First released in Issue 1. Derived from Issue 1 of the SVID.

47613 **Issue 6**

47614 Extensions beyond the ISO C standard are now marked.

47615 **NAME**

47616 towctrans — wide-character transliteration

47617 **SYNOPSIS**

47618
```
#include <wctype.h>
```

47619
```
wint_t towctrans(wint_t wc, wctrans_t desc);
```

47620 **DESCRIPTION**

47621 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47622 conflict between the requirements described here and the ISO C standard is unintentional. This
47623 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47624 The *towctrans*( ) function shall transliterate the wide-character code *wc* using the mapping
47625 described by *desc*. The current setting of the *LC_CTYPE* category should be the same as during
47626 CX the call to *wctrans*( ) that returned the value *desc*. If the value of *desc* is invalid (that is, not
47627 obtained by a call to *wctrans*( ) or *desc* is invalidated by a subsequent call to *setlocale*( ) that has
47628 affected category *LC_CTYPE*), the result is unspecified.

47629 An application wishing to check for error situations should set *errno* to 0 before calling
47630 *towctrans*( ). If *errno* is non-zero on return, an error has occurred.

47631 **RETURN VALUE**

47632 If successful, the *towctrans*( ) function shall return the mapped value of *wc* using the mapping
47633 described by *desc*. Otherwise, it shall return *wc* unchanged.

47634 **ERRORS**

47635 The *towctrans*( ) function may fail if:

47636 CX [EINVAL] *desc* contains an invalid transliteration descriptor.

47637 **EXAMPLES**

47638 None.

47639 **APPLICATION USAGE**

47640 The strings `"tolower"` and `"toupper"` are reserved for the standard mapping names. In the
47641 table below, the functions in the left column are equivalent to the functions in the right column.

47642
```
towlower(wc)    towctrans(wc, wctrans("tolower"))
```
47643
```
towupper(wc)    towctrans(wc, wctrans("toupper"))
```

47644 **RATIONALE**

47645 None.

47646 **FUTURE DIRECTIONS**

47647 None.

47648 **SEE ALSO**

47649 *towlower*( ), *towupper*( ), *wctrans*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
47650 **<wctype.h>**

47651 **CHANGE HISTORY**

47652 First released in Issue 5. Derived from ISO / IEC 9899: 1990 / Amendment 1: 1995 (E).

47653 **Issue 6**

47654 Extensions beyond the ISO C standard are now marked.

47655 **NAME**

47656 towlower — transliterate uppercase wide-character code to lowercase

47657 **SYNOPSIS**

47658 `#include <wctype.h>`

47659 `wint_t towlower(wint_t wc);`

47660 **DESCRIPTION**

47661 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47662 conflict between the requirements described here and the ISO C standard is unintentional. This
47663 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47664 The *towlower*( ) function has as a domain a type **wint_t**, the value of which the application shall
47665 ensure is a character representable as a **wchar_t**, and a wide-character code corresponding to a
47666 valid character in the current locale or the value of WEOF. If the argument has any other value,
47667 the behavior is undefined. If the argument of *towlower*( ) represents an uppercase wide-character
47668 code, and there exists a corresponding lowercase wide-character code (as defined by character
47669 type information in the program locale category *LC_CTYPE*), the result shall be the |
47670 corresponding lowercase wide-character code. All other arguments in the domain are returned |
47671 unchanged. |

47672 **RETURN VALUE**

47673 Upon successful completion, *towlower*( ) shall return the lowercase letter corresponding to the
47674 argument passed; otherwise, it shall return the argument unchanged.

47675 **ERRORS**

47676 No errors are defined.

47677 **EXAMPLES**

47678 None.

47679 **APPLICATION USAGE**

47680 None.

47681 **RATIONALE**

47682 None.

47683 **FUTURE DIRECTIONS**

47684 None.

47685 **SEE ALSO**

47686 *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the
47687 Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

47688 **CHANGE HISTORY**

47689 First released in Issue 4.

47690 **Issue 5**

47691 The following change has been made in this issue for alignment with
47692 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

47693 • The SYNOPSIS has been changed to indicate that this function and associated data types are
47694 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

47695 **Issue 6**

47696 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

47697 **NAME**

47698 towupper — transliterate lowercase wide-character code to uppercase

47699 **SYNOPSIS**

47700
```
#include <wctype.h>
```

47701
```
wint_t towupper(wint_t wc);
```

47702 **DESCRIPTION**

47703 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47704 conflict between the requirements described here and the ISO C standard is unintentional. This
47705 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47706 The *towupper*( ) function has as a domain a type **wint_t**, the value of which the application shall
47707 ensure is a character representable as a **wchar_t**, and a wide-character code corresponding to a
47708 valid character in the current locale or the value of WEOF. If the argument has any other value,
47709 the behavior is undefined. If the argument of *towupper*( ) represents a lowercase wide-character
47710 code, and there exists a corresponding uppercase wide-character code (as defined by character
47711 type information in the program locale category *LC_CTYPE*), the result shall be the |
47712 corresponding uppercase wide-character code. All other arguments in the domain are returned |
47713 unchanged. |

47714 **RETURN VALUE**

47715 Upon successful completion, *towupper*( ) shall return the uppercase letter corresponding to the
47716 argument passed. Otherwise, it shall return the argument unchanged.

47717 **ERRORS**

47718 No errors are defined.

47719 **EXAMPLES**

47720 None.

47721 **APPLICATION USAGE**

47722 None.

47723 **RATIONALE**

47724 None.

47725 **FUTURE DIRECTIONS**

47726 None.

47727 **SEE ALSO**

47728 *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the
47729 Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

47730 **CHANGE HISTORY**

47731 First released in Issue 4.

47732 **Issue 5**

47733 The following change has been made in this issue for alignment with
47734 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

47735 • The SYNOPSIS has been changed to indicate that this function and associated data types are
47736 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

47737 **Issue 6**

47738 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

47739 **NAME**

47740 trunc, truncf, truncl — round to truncated integer value

47741 **SYNOPSIS**

47742
```
#include <math.h>
```

47743
```
double trunc(double x);
```
47744
```
float truncf(float x);
```
47745
```
long double truncl(long double x);
```

47746 **DESCRIPTION**

47747 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47748 conflict between the requirements described here and the ISO C standard is unintentional. This
47749 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47750 These functions shall round their argument to the integer value, in floating format, nearest to but
47751 no larger in magnitude than the argument.

47752 **RETURN VALUE**

47753 Upon successful completion, these functions shall return the truncated integer value.

47754 MX If *x* is NaN, a NaN shall be returned.

47755 If *x* is ±0, or ±Inf, *x* shall be returned.

47756 **ERRORS**

47757 No errors are defined.

47758 **EXAMPLES**

47759 None.

47760 **APPLICATION USAGE**

47761 None.

47762 **RATIONALE**

47763 None.

47764 **FUTURE DIRECTIONS**

47765 None.

47766 **SEE ALSO**

47767 The Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

47768 **CHANGE HISTORY**

47769 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

47770 **NAME**

47771 truncate — truncate a file to a specified length

47772 **SYNOPSIS**

47773 XSI
```
#include <unistd.h>

int truncate(const char *path, off_t length);
```
47774

47775

47776 **DESCRIPTION**

47777 The *truncate*( ) function shall cause the regular file named by *path* to have a size which shall be
47778 equal to *length* bytes.

47779 If the file previously was larger than *length*, the extra data is discarded. If the file was previously
47780 shorter than *length*, its size is increased, and the extended area appears as if it were zero-filled.

47781 The application shall ensure that the process has write permission for the file.

47782 If the request would cause the file size to exceed the soft file size limit for the process, the
47783 request shall fail and the implementation shall generate the SIGXFSZ signal for the process.

47784 This function shall not modify the file offset for any open file descriptions associated with the
47785 file. Upon successful completion, if the file size is changed, this function shall mark for update
47786 the *st_ctime* and *st_mtime* fields of the file, and the S_ISUID and S_ISGID bits of the file mode
47787 may be cleared.

47788 **RETURN VALUE**

47789 Upon successful completion, *truncate*( ) shall return 0. Otherwise, –1 shall be returned, and *errno*
47790 set to indicate the error.

47791 **ERRORS**

47792 The *truncate*( ) function shall fail if:

47793 [EINTR] A signal was caught during execution.

47794 [EINVAL] The *length* argument was less than 0.

47795 [EFBIG] or [EINVAL]
47796 The *length* argument was greater than the maximum file size.

47797 [EIO] An I/O error occurred while reading from or writing to a file system.

47798 [EACCES] A component of the path prefix denies search permission, or write permission
47799 is denied on the file.

47800 [EISDIR] The named file is a directory.

47801 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
47802 argument.

47803 [ENAMETOOLONG]
47804 The length of the *path* argument exceeds {PATH_MAX} or a pathname |
47805 component is longer than {NAME_MAX}. |

47806 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

47807 [ENOTDIR] A component of the path prefix of *path* is not a directory.

47808 [EROFS] The named file resides on a read-only file system.

47809 The *truncate*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}. |

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Large File Summit extensions are added.

Issue 6

This reference page is split out from the *ftruncate*( ) reference page.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

47835 **NAME**

47836 truncf, truncl — round to truncated integer value

47837 **SYNOPSIS**

47838 `#include <math.h>`

47839 `float truncf(float x);`
47840 `long double truncl(long double x);`

47841 **DESCRIPTION**

47842 Refer to *trunc*( ).

47843 **NAME**

47844 tsearch — search a binary search tree

47845 **SYNOPSIS**

47846 XSI
```
#include <search.h>

void *tsearch(const void *key, void **rootp,
    int (*compar)(const void *, const void *));
```
47847
47848
47849

47850 **DESCRIPTION**

47851 Refer to *tdelete*( ).

47852 **NAME**

47853 ttyname, ttyname_r — find pathname of a terminal

47854 **SYNOPSIS**

47855 ```
#include <unistd.h>
```

47856 ```
char *ttyname(int fildes);
```

47857 TSF ```
int ttyname_r(int fildes, char *name, size_t namesize);
```

47858

47859 **DESCRIPTION**

47860 The *ttyname*( ) function shall return a pointer to a string containing a null-terminated pathname
47861 of the terminal associated with file descriptor *fildes*. The return value may point to static data
47862 whose content is overwritten by each call.

47863 The *ttyname*( ) function need not be reentrant. A function that is not required to be reentrant is
47864 not required to be thread-safe.

47865 TSF The *ttyname_r*( ) function shall store the null-terminated pathname of the terminal associated
47866 with the file descriptor *fildes* in the character array referenced by *name*. The array is *namesize*
47867 characters long and should have space for the name and the terminating null character. The
47868 maximum length of the terminal name shall be {TTY_NAME_MAX}.

47869 **RETURN VALUE**

47870 Upon successful completion, *ttyname*( ) shall return a pointer to a string. Otherwise, a null
47871 pointer shall be returned and *errno* set to indicate the error.

47872 TSF If successful, the *ttyname_r*( ) function shall return zero. Otherwise, an error number shall be
47873 returned to indicate the error.

47874 **ERRORS**

47875 The *ttyname*( ) function may fail if:

47876 [EBADF] The *fildes* argument is not a valid file descriptor.

47877 [ENOTTY] The *fildes* argument does not refer to a terminal.

47878 The *ttyname_r*( ) function may fail if:

47879 TSF [EBADF] The *fildes* argument is not a valid file descriptor.

47880 TSF [ENOTTY] The *fildes* argument does not refer to a terminal.

47881 TSF [ERANGE] The value of *namesize* is smaller than the length of the string to be returned
47882 including the terminating null character.

47883 **EXAMPLES**

47884 None.

47885 **APPLICATION USAGE**

47886 None.

47887 **RATIONALE**

47888 The term *terminal* is used instead of the historical term *terminal device* in order to avoid a
47889 reference to an undefined term.

47890 The thread-safe version places the terminal name in a user-supplied buffer and returns a non-
47891 zero value if it fails. The non-thread-safe version may return the name in a static data area that
47892 may be overwritten by each call.
```

47893 **FUTURE DIRECTIONS**

47894 None.

47895 **SEE ALSO**

47896 The Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

47897 **CHANGE HISTORY**

47898 First released in Issue 1. Derived from Issue 1 of the SVID.

47899 **Issue 5**

47900 The *ttyname_r*( ) function is included for alignment with the POSIX Threads Extension.

47901 A note indicating that the *ttyname*( ) function need not be reentrant is added to the
47902 DESCRIPTION.

47903 **Issue 6**

47904 The *ttyname_r*( ) function is marked as part of the Thread-Safe Functions option.

47905 The following new requirements on POSIX implementations derive from alignment with the
47906 Single UNIX Specification:

47907 - The statement that *errno* is set on error is added.

47908 - The [EBADF] and [ENOTTY] optional error conditions are added.

47909 **NAME**

47910 twalk — traverse a binary search tree

47911 **SYNOPSIS**

47912 XSI
```
#include <search.h>

void twalk(const void *root,
    void (*action)(const void *, VISIT, int ));
```
47915

47916 **DESCRIPTION**

47917 Refer to *tdelete*( ).

47918 **NAME**

47919 tzname — timezone strings

47920 **SYNOPSIS**

47921 CX
```
#include <time.h>

extern char *tzname[2];
```
47922

47923

47924 **DESCRIPTION**

47925 Refer to *tzset*( ).

47926 **NAME**

47927 daylight, timezone, tzname, tzset — set timezone conversion information

47928 **SYNOPSIS**

47929
```
#include <time.h>
```

47930 XSI
```
extern int daylight;
```
47931
```
extern long timezone;
```
47932 CX
```
extern char *tzname[2];
```
47933
```
void tzset(void);
```
47934

47935 **DESCRIPTION**

47936 The *tzset*( ) function shall use the value of the environment variable *TZ* to set time conversion
47937 information used by *ctime*( ), *localtime*( ), *mktime*( ), and *strftime*( ). If *TZ* is absent from the
47938 environment, implementation-defined default timezone information shall be used.

47939 The *tzset*( ) function shall set the external variable *tzname* as follows:

47940
```
tzname[0] = "std";
```
47941
```
tzname[1] = "dst";
```

47942 where *std* and *dst* are as described in the Base Definitions volume of IEEE Std 1003.1-200x,
47943 Chapter 8, Environment Variables.

47944 XSI The *tzset*( ) function also shall set the external variable *daylight* to 0 if Daylight Savings Time
47945 conversions should never be applied for the timezone in use; otherwise, non-zero. The external
47946 variable *timezone* shall be set to the difference, in seconds, between Coordinated Universal Time
47947 (UTC) and local standard time.

47948 **RETURN VALUE**

47949 The *tzset*( ) function shall not return a value.

47950 **ERRORS**

47951 No errors are defined.

47952 **EXAMPLES**

47953 Example TZ variables and their timezone differences are given in the table below:

47954

47955

TZ	*timezone*
EST5EDT	5*60*60
GMT0	0*60*60
JST-9	–9*60*60
MET-1MEST	–1*60*60
MST7MDT	7*60*60
PST8PDT	8*60*60

47962 **APPLICATION USAGE**

47963 None.

47964 **RATIONALE**

47965 None.

47966 **FUTURE DIRECTIONS**

47967 None.

47968 **SEE ALSO**

47969 *ctime*( ), *localtime*( ), *mktime*( ), *strftime*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
47970 **<time.h>**

47971 **CHANGE HISTORY**

47972 First released in Issue 1. Derived from Issue 1 of the SVID. |

47973 **Issue 6** |

47974 The example is corrected. |

47975 **NAME**

47976 ualarm — set the interval timer

47977 **SYNOPSIS**

47978 OB XSI
```
#include <unistd.h>
```

47979
```
useconds_t ualarm(useconds_t useconds, useconds_t interval);
```

47980

47981 **DESCRIPTION**

47982 The *ualarm*( ) function shall cause the SIGALRM signal to be generated for the calling process
47983 after the number of realtime microseconds specified by the *useconds* argument has elapsed.
47984 When the *interval* argument is non-zero, repeated timeout notification occurs with a period in
47985 microseconds specified by the *interval* argument. If the notification signal, SIGALRM, is not
47986 caught or ignored, the calling process is terminated.

47987 Implementations may place limitations on the granularity of timer values. For each interval
47988 timer, if the requested timer value requires a finer granularity than the implementation supports,
47989 the actual timer value shall be rounded up to the next supported value.

47990 Interactions between *ualarm*( ) and any of the following are unspecified:

47991 *alarm*( )
47992 *nanosleep*( )
47993 *setitimer*( )
47994 *timer_create*( )
47995 *timer_delete*( )
47996 *timer_getoverrun*( )
47997 *timer_gettime*( )
47998 *timer_settime*( )
47999 *sleep*( )

48000 **RETURN VALUE**

48001 The *ualarm*( ) function shall return the number of microseconds remaining from the previous
48002 *ualarm*( ) call. If no timeouts are pending or if *ualarm*( ) has not previously been called, *ualarm*( )
48003 shall return 0.

48004 **ERRORS**

48005 No errors are defined.

48006 **EXAMPLES**

48007 None.

48008 **APPLICATION USAGE**

48009 Applications are recommended to use *nanosleep*( ) if the Timers option is supported, or
48010 *setitimer*( ), *timer_create*( ), *timer_delete*( ), *timer_getoverrun*( ), *timer_gettime*( ), or *timer_settime*( )
48011 instead of this function.

48012 **RATIONALE**

48013 None.

48014 **FUTURE DIRECTIONS**

48015 None.

48016 **SEE ALSO**

48017 *alarm*( ), *nanosleep*( ), *setitimer*( ), *sleep*( ), *timer_create*( ), *timer_delete*( ), *timer_getoverrun*( ), the Base
48018 Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

48019 **CHANGE HISTORY**

48020 First released in Issue 4, Version 2.

48021 **Issue 5**

48022 Moved from X/OPEN UNIX extension to BASE.

48023 **Issue 6**

48024 This function is marked obsolescent.

48025 **NAME**

48026 ulimit — get and set process limits

48027 **SYNOPSIS**

48028 XSI
```
#include <ulimit.h>
```

48029
```
long ulimit(int cmd, ...);
```

48030

48031 **DESCRIPTION**

48032 The *ulimit*( ) function shall control process limits. The process limits that can be controlled by
48033 this function include the maximum size of a single file that can be written (this is equivalent to
48034 using *setrlimit*( ) with RLIMIT_FSIZE). The *cmd* values, defined in **<ulimit.h>** include:

48035 UL_GETFSIZE Return the file size limit (RLIMIT_FSIZE) of the process. The limit shall be in
48036 units of 512-byte blocks and shall be inherited by child processes. Files of any
48037 size can be read. The return value shall be the integer part of the soft file size
48038 limit divided by 512. If the result cannot be represented as a **long**, the result is
48039 unspecified.

48040 UL_SETFSIZE Set the file size limit for output operations of the process to the value of the
48041 second argument, taken as a **long**, multiplied by 512. If the result would
48042 overflow an **rlim_t**, the actual value set is unspecified. Any process may
48043 decrease its own limit, but only a process with appropriate privileges may
48044 increase the limit. The return value shall be the integer part of the new file size
48045 limit divided by 512.

48046 The *ulimit*( ) function shall not change the setting of *errno* if successful.

48047 As all return values are permissible in a successful situation, an application wishing to check for
48048 error situations should set *errno* to 0, then call *ulimit*( ), and, if it returns –1, check to see if *errno* is
48049 non-zero.

48050 **RETURN VALUE**

48051 Upon successful completion, *ulimit*( ) shall return the value of the requested limit. Otherwise, –1
48052 shall be returned and *errno* set to indicate the error.

48053 **ERRORS**

48054 The *ulimit*( ) function shall fail and the limit shall be unchanged if:

48055 [EINVAL] The *cmd* argument is not valid.

48056 [EPERM] A process not having appropriate privileges attempts to increase its file size
48057 limit.

48058 **EXAMPLES**

48059 None.

48060 **APPLICATION USAGE**

48061 None.

48062 **RATIONALE**

48063 None.

48064 **FUTURE DIRECTIONS**

48065 None.

48066 **SEE ALSO**

48067 *getrlimit*( ), *setrlimit*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ulimit.h>**

48068 **CHANGE HISTORY**

48069 First released in Issue 1. Derived from Issue 1 of the SVID.

48070 **Issue 5**

48071 In the description of UL_SETFSIZE, the text is corrected to refer to **rlim_t** rather than the
48072 spurious **rlimit_t**.

48073 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

48074 **NAME**

48075 umask — set and get file mode creation mask

48076 **SYNOPSIS**

48077 `#include <sys/stat.h>`

48078 `mode_t umask(mode_t cmask);`

48079 **DESCRIPTION**

48080 The *umask*( ) function shall set the process' file mode creation mask to *cmask* and return the
48081 previous value of the mask. Only the file permission bits of *cmask* (see **<sys/stat.h>**) are used; the
48082 meaning of the other bits is implementation-defined.

48083 The process' file mode creation mask is used during *open*( ), *creat*( ), *mkdir*( ), and *mkfifo*( ) to turn
48084 off permission bits in the *mode* argument supplied. Bit positions that are set in *cmask* are cleared
48085 in the mode of the created file.

48086 **RETURN VALUE**

48087 The file permission bits in the value returned by *umask*( ) shall be the previous value of the file
48088 mode creation mask. The state of any other bits in that value is unspecified, except that a
48089 subsequent call to *umask*( ) with the returned value as *cmask* shall leave the state of the mask the
48090 same as its state before the first call, including any unspecified use of those bits.

48091 **ERRORS**

48092 No errors are defined.

48093 **EXAMPLES**

48094 None.

48095 **APPLICATION USAGE**

48096 None.

48097 **RATIONALE**

48098 Unsigned argument and return types for *umask*( ) were proposed. The return type and the
48099 argument were both changed to **mode_t**.

48100 Historical implementations have made use of additional bits in *cmask* for their implementation-
48101 defined purposes. The addition of the text that the meaning of other bits of the field is
48102 implementation-defined permits these implementations to conform to this volume of
48103 IEEE Std 1003.1-200x.

48104 **FUTURE DIRECTIONS**

48105 None.

48106 **SEE ALSO**

48107 *creat*( ), *mkdir*( ), *mkfifo*( ), *open*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
48108 **<sys/stat.h>**, **<sys/types.h>**

48109 **CHANGE HISTORY**

48110 First released in Issue 1. Derived from Issue 1 of the SVID.

48111 **Issue 6**

48112 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed. |

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

48118 **NAME**

48119 uname — get name of current system

48120 **SYNOPSIS**

48121 `#include <sys/utsname.h>`

48122 `int uname(struct utsname *`*`name`*`);`

48123 **DESCRIPTION**

48124 The *uname*( ) function shall store information identifying the current system in the structure
48125 pointed to by *name*.

48126 The *uname*( ) function uses the **utsname** structure defined in **<sys/utsname.h>**.

48127 The *uname*( ) function shall return a string naming the current system in the character array
48128 *sysname*. Similarly, *nodename* shall contain the name of this node within an implementation-
48129 defined communications network. The arrays *release* and *version* shall further identify the
48130 operating system. The array *machine* shall contain a name that identifies the hardware that the
48131 system is running on.

48132 The format of each member is implementation-defined.

48133 **RETURN VALUE**

48134 Upon successful completion, a non-negative value shall be returned. Otherwise, –1 shall be
48135 returned and *errno* set to indicate the error.

48136 **ERRORS**

48137 No errors are defined.

48138 **EXAMPLES**

48139 None.

48140 **APPLICATION USAGE**

48141 The inclusion of the *nodename* member in this structure does not imply that it is sufficient
48142 information for interfacing to communications networks.

48143 **RATIONALE**

48144 The values of the structure members are not constrained to have any relation to the version of
48145 this volume of IEEE Std 1003.1-200x implemented in the operating system. An application
48146 should instead depend on _POSIX_VERSION and related constants defined in **<unistd.h>**.

48147 This volume of IEEE Std 1003.1-200x does not define the sizes of the members of the structure
48148 and permits them to be of different sizes, although most implementations define them all to be
48149 the same size: eight bytes plus one byte for the string terminator. That size for *nodename* is not
48150 enough for use with many networks.

48151 The *uname*( ) function originated in System III, System V, and related implementations, and it
48152 does not exist in Version 7 or 4.3 BSD. The values it returns are set at system compile time in
48153 those historical implementations.

48154 4.3 BSD has *gethostname*( ) and *gethostid*( ), which return a symbolic name and a numeric value,
48155 respectively. There are related *sethostname*( ) and *sethostid*( ) functions that are used to set the
48156 values the other two functions return. The former functions are included in this specification, the
48157 latter are not.

48158 **FUTURE DIRECTIONS**

48159 None.

48160 **SEE ALSO**

48161 The Base Definitions volume of IEEE Std 1003.1-200x, **<sys/utsname.h>**

48162 **CHANGE HISTORY**

48163 First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

ungetc — push byte back into input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc(int c, FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *ungetc*( ) function shall push the byte specified by *c* (converted to an **unsigned char**) back onto the input stream pointed to by *stream*. The pushed-back bytes shall be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file-positioning function (*fseek*( ), *fsetpos*( ), or *rewind*( )) shall discard any pushed-back bytes for the stream. The external storage corresponding to the stream shall be unchanged.

One byte of push-back shall be provided. If *ungetc*( ) is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the operation may fail.

If the value of *c* equals that of the macro EOF, the operation shall fail and the input stream shall be left unchanged.

A successful call to *ungetc*( ) shall clear the end-of-file indicator for the stream. The value of the file-position indicator for the stream after reading or discarding all pushed-back bytes shall be the same as it was before the bytes were pushed back. The file-position indicator is decremented by each successful call to *ungetc*( ); if its value was 0 before a call, its value is unspecified after the call.

RETURN VALUE

Upon successful completion, *ungetc*( ) shall return the byte pushed back after conversion. Otherwise, it shall return EOF.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fseek( ), *getc*( ), *fsetpos*( ), *read*( ), *rewind*( ), *setbuf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

48206 **NAME**

48207 ungetwc — push wide-character code back into input stream

48208 **SYNOPSIS**

48209 `#include <stdio.h>`
48210 `#include <wchar.h>`

48211 `wint_t ungetwc(wint_t wc, FILE *stream);`

48212 **DESCRIPTION**

48213 CX The functionality described on this reference page is aligned with the ISO C standard. Any
48214 conflict between the requirements described here and the ISO C standard is unintentional. This
48215 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

48216 The *ungetwc*( ) function shall push the character corresponding to the wide-character code
48217 specified by *wc* back onto the input stream pointed to by *stream*. The pushed-back characters
48218 shall be returned by subsequent reads on that stream in the reverse order of their pushing. A
48219 successful intervening call (with the stream pointed to by *stream*) to a file-positioning function
48220 (*fseek*( ), *fsetpos*( ), or *rewind*( )) discards any pushed-back characters for the stream. The external
48221 storage corresponding to the stream is unchanged.

48222 At least one character of push-back shall be provided. If *ungetwc*( ) is called too many times on |
48223 the same stream without an intervening read or file-positioning operation on that stream, the
48224 operation may fail.

48225 If the value of *wc* equals that of the macro WEOF, the operation shall fail and the input stream |
48226 shall be left unchanged. |

48227 A successful call to *ungetwc*( ) shall clear the end-of-file indicator for the stream. The value of the
48228 file-position indicator for the stream after reading or discarding all pushed-back characters shall
48229 be the same as it was before the characters were pushed back. The file-position indicator is
48230 decremented (by one or more) by each successful call to *ungetwc*( ); if its value was 0 before a |
48231 call, its value is unspecified after the call. |

48232 **RETURN VALUE**

48233 Upon successful completion, *ungetwc*( ) shall return the wide-character code corresponding to
48234 the pushed-back character. Otherwise, it shall return WEOF.

48235 **ERRORS**

48236 The *ungetwc*( ) function may fail if:

48237 CX [EILSEQ] An invalid character sequence is detected, or a wide-character code does not |
48238 correspond to a valid character. |

48239 **EXAMPLES**

48240 None.

48241 **APPLICATION USAGE**

48242 None.

48243 **RATIONALE**

48244 None.

48245 **FUTURE DIRECTIONS**

48246 None.

48247 **SEE ALSO**

48248 *fseek*( ), *fsetpos*( ), *read*( ), *rewind*( ), *setbuf*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
48249 **<stdio.h>**, **<wchar.h>**

48250 **CHANGE HISTORY**

48251 First released in Issue 4. Derived from the MSE working draft.

48252 **Issue 5**

48253 The Optional Header (OH) marking is removed from **<stdio.h>**. |

48254 **Issue 6** |

48255 The [EILSEQ] optional error condition is marked CX. |

48256 **NAME**

48257 unlink — remove a directory entry

48258 **SYNOPSIS**

48259 `#include <unistd.h>`

48260 `int unlink(const char *path);`

48261 **DESCRIPTION**

48262 The *unlink*( ) function shall remove a link to a file. If *path* names a symbolic link, *unlink*( ) shall |
48263 remove the symbolic link named by *path* and shall not affect any file or directory named by the |
48264 contents of the symbolic link. Otherwise, *unlink*( ) shall remove the link named by the pathname |
48265 pointed to by *path* and shall decrement the link count of the file referenced by the link. |

48266 When the file's link count becomes 0 and no process has the file open, the space occupied by the
48267 file shall be freed and the file shall no longer be accessible. If one or more processes have the file
48268 open when the last link is removed, the link shall be removed before *unlink*( ) returns, but the
48269 removal of the file contents shall be postponed until all references to the file are closed.

48270 The *path* argument shall not name a directory unless the process has appropriate privileges and |
48271 the implementation supports using *unlink*( ) on directories.

48272 Upon successful completion, *unlink*( ) shall mark for update the *st_ctime* and *st_mtime* fields of
48273 the parent directory. Also, if the file's link count is not 0, the *st_ctime* field of the file shall be
48274 marked for update.

48275 **RETURN VALUE**

48276 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
48277 indicate the error. If –1 is returned, the named file shall not be changed.

48278 **ERRORS**

48279 The *unlink*( ) function shall fail and shall not unlink the file if:

48280 [EACCES] Search permission is denied for a component of the path prefix, or write
48281 permission is denied on the directory containing the directory entry to be
48282 removed.

48283 [EBUSY] The file named by the *path* argument cannot be unlinked because it is being
48284 used by the system or another process and the implementation considers this
48285 an error.

48286 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
48287 argument.

48288 [ENAMETOOLONG]
48289 The length of the *path* argument exceeds {PATH_MAX} or a pathname |
48290 component is longer than {NAME_MAX}. |

48291 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

48292 [ENOTDIR] A component of the path prefix is not a directory.

48293 [EPERM] The file named by *path* is a directory, and either the calling process does not
48294 have appropriate privileges, or the implementation prohibits using *unlink*( )
48295 on directories.

48296 XSI [EPERM] or [EACCES]
48297 The S_ISVTX flag is set on the directory containing the file referred to by the
48298 *path* argument and the caller is not the file owner, nor is the caller the
48299 directory owner, nor does the caller have appropriate privileges.

[EROFS] The directory entry to be unlinked is part of a read-only file system.

The *unlink*( ) function may fail and not unlink the file if:

XSI [EBUSY] The file named by *path* is a named STREAM.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

[ETXTBSY] The entry to be unlinked is the last directory entry to a pure procedure (shared text) file that is being executed.

EXAMPLES

Removing a Link to a File

The following example shows how to remove a link to a file named **/home/cnd/mod1** by removing the entry named **/modules/pass1**.

```
#include <unistd.h>

char *path = "/modules/pass1";
int   status;
...
status = unlink(path);
```

Checking for an Error

The following example fragment creates a temporary password lock file named **LOCKFILE**, which is defined as **/etc/ptmp**, and gets a file descriptor for it. If the file cannot be opened for writing, *unlink*( ) is used to remove the link between the file descriptor and **LOCKFILE**.

```
#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>

#define LOCKFILE "/etc/ptmp"

int pfd;  /* Integer for file descriptor returned by open call. */
FILE *fpfd;  /* File pointer for use in putpwent(). */
...
/* Open password Lock file. If it exists, this is an error. */
if ((pfd = open(LOCKFILE, O_WRONLY| O_CREAT | O_EXCL, S_IRUSR
    | S_IWUSR | S_IRGRP | S_IROTH)) == -1)  {
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}

/* Lock file created, proceed with fdopen of lock file so that
   putpwent() can be used.
 */
if ((fpfd = fdopen(pfd, "w")) == NULL) {
```

```
        close(pfd);
        unlink(LOCKFILE);
        exit(1);
    }
```

Replacing Files

The following example fragment uses *unlink*() to discard links to files, so that they can be replaced with new versions of the files. The first call remove the link to **LOCKFILE** if an error occurs. Successive calls remove the links to **SAVEFILE** and **PASSWDFILE** so that new links can be created, then removes the link to **LOCKFILE** when it is no longer needed.

```
#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
#define SAVEFILE "/etc/opasswd"
...
/* If no change was made, assume error and leave passwd unchanged. */
if (!valid_change) {
    fprintf(stderr, "Could not change password for user %s\n", user);
    unlink(LOCKFILE);
    exit(1);
}

/* Change permissions on new password file. */
chmod(LOCKFILE, S_IRUSR | S_IRGRP | S_IROTH);

/* Remove saved password file. */
unlink(SAVEFILE);

/* Save current password file. */
link(PASSWDFILE, SAVEFILE);

/* Remove current password file. */
unlink(PASSWDFILE);

/* Save new password file as current password file. */
link(LOCKFILE,PASSWDFILE);

/* Remove lock file. */
unlink(LOCKFILE);

exit(0);
```

APPLICATION USAGE

Applications should use *rmdir*() to remove a directory.

RATIONALE

Unlinking a directory is restricted to the superuser in many historical implementations for reasons given in *link*() (see also *rename*()).

The meaning of [EBUSY] in historical implementations is ‘‘mount point busy’’. Since this volume of IEEE Std 1003.1-200x does not cover the system administration concepts of mounting and unmounting, the description of the error was changed to ‘‘resource busy’’. (This meaning is used by some device drivers when a second process tries to open an exclusive use device.) The wording is also intended to allow implementations to refuse to remove a directory if it is the root or current working directory of any process.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *link*( ), *remove*( ), *rmdir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The [EBUSY] error is added to the ‘‘may fail’’ part of the ERRORS section.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- In the DESCRIPTION, the effect is specified if *path* specifies a symbolic link.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.
- The [ETXTBSY] optional error condition is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

48411 **NAME**

48412 unlockpt — unlock a pseudo-terminal master/slave pair

48413 **SYNOPSIS**

48414 XSI
```
#include <stdlib.h>

int unlockpt(int fildes);
```
48415

48416

48417 **DESCRIPTION**

48418 The *unlockpt*( ) function shall unlock the slave pseudo-terminal device associated with the
48419 master to which *fildes* refers.

48420 Conforming applications shall ensure that they call *unlockpt*( ) before opening the slave side of a |
48421 pseudo-terminal device.

48422 **RETURN VALUE**

48423 Upon successful completion, *unlockpt*( ) shall return 0. Otherwise, it shall return –1 and set *errno*
48424 to indicate the error.

48425 **ERRORS**

48426 The *unlockpt*( ) function may fail if:

48427 [EBADF] The *fildes* argument is not a file descriptor open for writing.

48428 [EINVAL] The *fildes* argument is not associated with a master pseudo-terminal device.

48429 **EXAMPLES**

48430 None.

48431 **APPLICATION USAGE**

48432 None.

48433 **RATIONALE**

48434 None.

48435 **FUTURE DIRECTIONS**

48436 None.

48437 **SEE ALSO**

48438 *grantpt*( ), *open*( ), *ptsname*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

48439 **CHANGE HISTORY**

48440 First released in Issue 4, Version 2.

48441 **Issue 5**

48442 Moved from X/OPEN UNIX extension to BASE.

48443 **Issue 6**

48444 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

unsetenv — remove environment variable

SYNOPSIS

CX
```
#include <stdlib.h>

int unsetenv(const char *name);
```

DESCRIPTION

The *unsetenv*( ) function shall remove an environment variable from the environment of the calling process. The *name* argument points to a string, which is the name of the variable to be removed. The named argument shall not contain an '=' character. If the named variable does not exist in the current environment, the environment shall be unchanged and the function is considered to have completed successfully.

If the application modifies *environ* or the pointers to which it points, the behavior of *unsetenv*( ) is undefined. The *unsetenv*( ) function shall update the list of pointers to which *environ* points.

The *unsetenv*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, zero shall be returned. Otherwise, –1 shall be returned, *errno* set to indicate the error, and the environment shall be unchanged.

ERRORS

The *unsetenv*( ) function shall fail if:

[EINVAL] The *name* argument is a null pointer, points to an empty string, or points to a string containing an '=' character.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to the RATIONALE section in *setenv*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

getenv( ), *setenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

NAME

usleep — suspend execution for an interval

SYNOPSIS

OB XSI
```
#include <unistd.h>

int usleep(useconds_t useconds);
```

DESCRIPTION

The *usleep*( ) function shall cause the calling thread to be suspended from execution until either the number of realtime microseconds specified by the argument *useconds* has elapsed or a signal is delivered to the calling thread and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested due to the scheduling of other activity by the system.

The *useconds* argument shall be less than one million. If the value of *useconds* is 0, then the call has no effect. |

If a SIGALRM signal is generated for the calling process during execution of *usleep*( ) and if the SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether *usleep*( ) returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also unspecified whether it remains pending after *usleep*( ) returns or it is discarded.

If a SIGALRM signal is generated for the calling process during execution of *usleep*( ), except as a result of a prior call to *alarm*( ), and if the SIGALRM signal is not being ignored or blocked from delivery, it is unspecified whether that signal has any effect other than causing *usleep*( ) to return.

If a signal-catching function interrupts *usleep*( ) and examines or changes either the time a SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or whether the SIGALRM signal is blocked from delivery, the results are unspecified.

If a signal-catching function interrupts *usleep*( ) and calls *siglongjmp*( ) or *longjmp*( ) to restore an environment saved prior to the *usleep*( ) call, the action associated with the SIGALRM signal and the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also unspecified whether the SIGALRM signal is blocked, unless the process' signal mask is restored as part of the environment.

Implementations may place limitations on the granularity of timer values. For each interval timer, if the requested timer value requires a finer granularity than the implementation supports, the actual timer value shall be rounded up to the next supported value.

Interactions between *usleep*( ) and any of the following are unspecified:

nanosleep( )
setitimer( )
timer_create( )
timer_delete( )
timer_getoverrun( )
timer_gettime( )
timer_settime( )
ualarm( )
sleep( )

48523 **RETURN VALUE**

48524 Upon successful completion, *usleep*( ) shall return 0; otherwise, it shall return –1 and set *errno* to
48525 indicate the error.

48526 **ERRORS**

48527 The *usleep*( ) function may fail if:

48528 [EINVAL] The time interval specified one million or more microseconds.

48529 **EXAMPLES**

48530 None.

48531 **APPLICATION USAGE**

48532 Applications are recommended to use *nanosleep*( ) if the Timers option is supported, or
48533 *setitimer*( ), *timer_create*( ), *timer_delete*( ), *timer_getoverrun*( ), *timer_gettime*( ), or *timer_settime*( )
48534 instead of this function.

48535 **RATIONALE**

48536 None.

48537 **FUTURE DIRECTIONS**

48538 None.

48539 **SEE ALSO**

48540 *alarm*( ), *getitimer*( ), *nanosleep*( ), *sigaction*( ), *sleep*( ), *timer_create*( ), *timer_delete*( ),
48541 *timer_getoverrun*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

48542 **CHANGE HISTORY**

48543 First released in Issue 4, Version 2.

48544 **Issue 5**

48545 Moved from X/OPEN UNIX extension to BASE.

48546 The DESCRIPTION is changed to indicate that timers are now thread-based rather than
48547 process-based.

48548 **Issue 6**

48549 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

48550 This function is marked obsolescent.

48551 **NAME**

48552 utime — set file access and modification times

48553 **SYNOPSIS**

48554 `#include <utime.h>`

48555 `int utime(const char *path, const struct utimbuf *times);`

48556 **DESCRIPTION**

48557 The *utime*( ) function shall set the access and modification times of the file named by the *path*
48558 argument.

48559 If *times* is a null pointer, the access and modification times of the file shall be set to the current |
48560 time. The effective user ID of the process shall match the owner of the file, or the process has |
48561 write permission to the file or has appropriate privileges, to use *utime*( ) in this manner. |

48562 If *times* is not a null pointer, *times* shall be interpreted as a pointer to a **utimbuf** structure and the |
48563 access and modification times shall be set to the values contained in the designated structure. |
48564 Only a process with effective user ID equal to the user ID of the file or a process with |
48565 appropriate privileges may use *utime*( ) this way. |

48566 The **utimbuf** structure is defined in the **<utime.h>** header. The times in the structure **utimbuf** |
48567 are measured in seconds since the Epoch.

48568 Upon successful completion, *utime*( ) shall mark the time of the last file status change, *st_ctime*,
48569 to be updated; see **<sys/stat.h>**.

48570 **RETURN VALUE**

48571 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* shall
48572 be set to indicate the error, and the file times shall not be affected.

48573 **ERRORS**

48574 The *utime*( ) function shall fail if:

48575 [EACCES] Search permission is denied by a component of the path prefix; or the *times*
48576 argument is a null pointer and the effective user ID of the process does not
48577 match the owner of the file, the process does not have write permission for the
48578 file, and the process does not have appropriate privileges.

48579 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
48580 argument.

48581 [ENAMETOOLONG]
48582 The length of the *path* argument exceeds {PATH_MAX} or a pathname |
48583 component is longer than {NAME_MAX}. |

48584 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

48585 [ENOTDIR] A component of the path prefix is not a directory.

48586 [EPERM] The *times* argument is not a null pointer and the calling process' effective user
48587 ID does not match the owner of the file and the calling process does not have
48588 the appropriate privileges.

48589 [EROFS] The file system containing the file is read-only.

48590 The *utime*( ) function may fail if:

48591 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
48592 resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *actime* structure member must be present so that an application may set it, even though an implementation may ignore it and not change the access time on the file. If an application intends to leave one of the times of a file unchanged while changing the other, it should use *stat*( ) to retrieve the file's *st_atime* and *st_mtime* parameters, set *actime* and *modtime* in the buffer, and change one of them before making the *utime*( ) call.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<utime.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

48624 **NAME**

48625 utimes — set file access and modification times (**LEGACY**)

48626 **SYNOPSIS**

48627 XSI
```
#include <sys/time.h>

int utimes(const char *path, const struct timeval times[2]);
```
48628
48629

48630 **DESCRIPTION**

48631 The *utimes*( ) function shall set the access and modification times of the file pointed to by the *path*
48632 argument to the value of the *times* argument. The *utimes*( ) function allows time specifications
48633 accurate to the microsecond.

48634 For *utimes*( ), the *times* argument is an array of **timeval** structures. The first array member
48635 represents the date and time of last access, and the second member represents the date and time
48636 of last modification. The times in the **timeval** structure are measured in seconds and
48637 microseconds since the Epoch, although rounding toward the nearest second may occur.

48638 If the *times* argument is a null pointer, the access and modification times of the file shall be set to |
48639 the current time. The effective user ID of the process shall match the owner of the file, or has |
48640 write access to the file or appropriate privileges to use this call in this manner. Upon completion, |
48641 *utimes*( ) shall mark the time of the last file status change, *st_ctime*, for update.

48642 **RETURN VALUE**

48643 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* shall
48644 be set to indicate the error, and the file times shall not be affected.

48645 **ERRORS**

48646 The *utimes*( ) function shall fail if:

48647 [EACCES] Search permission is denied by a component of the path prefix; or the *times*
48648 argument is a null pointer and the effective user ID of the process does not
48649 match the owner of the file and write access is denied.

48650 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
48651 argument.

48652 [ENAMETOOLONG]
48653 The length of the *path* argument exceeds {PATH_MAX} or a pathname |
48654 component is longer than {NAME_MAX}. |

48655 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

48656 [ENOTDIR] A component of the path prefix is not a directory.

48657 [EPERM] The *times* argument is not a null pointer and the calling process' effective user
48658 ID has write access to the file but does not match the owner of the file and the
48659 calling process does not have the appropriate privileges.

48660 [EROFS] The file system containing the file is read-only.

48661 The *utimes*( ) function may fail if:

48662 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
48663 resolution of the *path* argument.

48664 [ENAMETOOLONG]
48665 Pathname resolution of a symbolic link produced an intermediate result |
48666 whose length exceeds {PATH_MAX}.

EXAMPLES

None.

APPLICATION USAGE

For applications portability, the *utime*() function should be used to set file access and modification times instead of *utimes*().

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<sys/time.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

48687 **NAME**

48688 va_arg, va_copy, va_end, va_start — handle variable argument list

48689 **SYNOPSIS**

48690
```
#include <stdarg.h>
```

48691
```
type va_arg(va_list ap, type);
```
48692
```
void va_copy(va_list dest, va_list src);
```
48693
```
void va_end(va_list ap);
```
48694
```
void va_start(va_list ap, argN);
```

48695 **DESCRIPTION**

48696 Refer to the Base Definitions volume of IEEE Std 1003.1-200x, **<stdarg.h>**.

48697 **NAME**

48698 vfork — create new process; share virtual memory

48699 **SYNOPSIS**

48700 OB XSI
```
#include <unistd.h>

pid_t vfork(void);
```
48701

48702

48703 **DESCRIPTION**

48704 The *vfork*( ) function shall be equivalent to *fork*( ), except that the behavior is undefined if the |
48705 process created by *vfork*( ) either modifies any data other than a variable of type **pid_t** used to
48706 store the return value from *vfork*( ), or returns from the function in which *vfork*( ) was called, or
48707 calls any other function before successfully calling *_exit*( ) or one of the *exec* family of functions.

48708 **RETURN VALUE**

48709 Upon successful completion, *vfork*( ) shall return 0 to the child process and return the process ID
48710 of the child process to the parent process. Otherwise, –1 shall be returned to the parent, no child
48711 process shall be created, and *errno* shall be set to indicate the error.

48712 **ERRORS**

48713 The *vfork*( ) function shall fail if:

48714 [EAGAIN] The system-wide limit on the total number of processes under execution
48715 would be exceeded, or the system-imposed limit on the total number of
48716 processes under execution by a single user would be exceeded.

48717 [ENOMEM] There is insufficient swap space for the new process.

48718 **EXAMPLES**

48719 None.

48720 **APPLICATION USAGE**

48721 Conforming applications are recommended not to depend on *vfork*( ), but to use *fork*( ) instead. |
48722 The *vfork*( ) function may be withdrawn in a future version.

48723 On some implementations, *vfork*( ) is equivalent to *fork*( ). |

48724 The *vfork*( ) function differs from *fork*( ) only in that the child process can share code and data
48725 with the calling process (parent process). This speeds cloning activity significantly at a risk to
48726 the integrity of the parent process if *vfork*( ) is misused.

48727 The use of *vfork*( ) for any purpose except as a prelude to an immediate call to a function from
48728 the *exec* family, or to *_exit*( ), is not advised.

48729 The *vfork*( ) function can be used to create new processes without fully copying the address
48730 space of the old process. If a forked process is simply going to call *exec*, the data space copied
48731 from the parent to the child by *fork*( ) is not used. This is particularly inefficient in a paged
48732 environment, making *vfork*( ) particularly useful. Depending upon the size of the parent's data
48733 space, *vfork*( ) can give a significant performance improvement over *fork*( ).

48734 The *vfork*( ) function can normally be used just like *fork*( ). It does not work, however, to return
48735 while running in the child's context from the caller of *vfork*( ) since the eventual return from
48736 *vfork*( ) would then return to a no longer existent stack frame. Care should be taken, also, to call
48737 *_exit*( ) rather than *exit*( ) if *exec* cannot be used, since *exit*( ) flushes and closes standard I/O
48738 channels, thereby damaging the parent process' standard I/O data structures. (Even with *fork*( ),
48739 it is wrong to call *exit*( ), since buffered data would then be flushed twice.)

48740 If signal handlers are invoked in the child process after *vfork*( ), they must follow the same rules
48741 as other code in the child process.

48742 **RATIONALE**

48743 None.

48744 **FUTURE DIRECTIONS**

48745 This function may be withdrawn in a future version.

48746 **SEE ALSO**

48747 *exec*, *exit*( ), *fork*( ), *wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

48748 **CHANGE HISTORY**

48749 First released in Issue 4, Version 2.

48750 **Issue 5**

48751 Moved from X/OPEN UNIX extension to BASE.

48752 **Issue 6**

48753 Marked obsolescent.

NAME

vfprintf, vprintf, vsnprintf, vsprintf — format output of a stdarg argument list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>

int vfprintf(FILE *restrict stream, const char *restrict format,
    va_list ap);
int vprintf(const char *restrict format, va_list ap);
int vsnprintf(char *restrict s, size_t n, const char *restrict format,
    va_list ap);
int vsprintf(char *restrict s, const char *restrict format, va_list ap);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *vprintf*(), *vfprintf*(), *vsnprintf*(), and *vsprintf*() functions shall be equivalent to *printf*(), *fprintf*(), *snprintf*(), and *sprintf*() respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by **<stdarg.h>**.

These functions shall not invoke the *va_end* macro. As these functions invoke the *va_arg* macro, the value of *ap* after the return is unspecified.

RETURN VALUE

Refer to *fprintf*().

ERRORS

Refer to *fprintf*().

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions should call *va_end*(*ap*) afterwards to clean up.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fprintf(), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdarg.h>**, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The *vsnprintf*() function is added.

Issue 6

The *vfprintf*(), *vprintf*(), *vsnprintf*(), and *vsprintf*() functions are updated for alignment with the ISO / IEC 9899: 1999 standard.

48795 **NAME**

48796 vfscanf, vscanf, vsscanf — format input of a stdarg list

48797 **SYNOPSIS**

48798 `#include <stdarg.h>`
48799 `#include <stdio.h>`

48800 `int vfscanf(FILE *restrict stream, const char *restrict format,`
48801 `    va_list arg);`
48802 `int vscanf(const char *restrict format, va_list arg);`
48803 `int vsscanf(const char *restrict s, const char *restrict format,`
48804 `    va_list arg);`

48805 **DESCRIPTION**

48806 CX The functionality described on this reference page is aligned with the ISO C standard. Any
48807 conflict between the requirements described here and the ISO C standard is unintentional. This
48808 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

48809 The *vscanf*( ), *vfscanf*( ), and *vsscanf*( ) functions shall be equivalent to the *scanf*( ), *fscanf*( ), and |
48810 *sscanf*( ) functions, respectively, except that instead of being called with a variable number of
48811 arguments, they are called with an argument list as defined in the **<stdarg.h>** header. These |
48812 functions shall not invoke the *va_end* macro. As these functions invoke the *va_arg* macro, the |
48813 value of *ap* after the return is unspecified. |

48814 **RETURN VALUE**

48815 Refer to *fscanf*( ).

48816 **ERRORS**

48817 Refer to *fscanf*( ).

48818 **EXAMPLES**

48819 None.

48820 **APPLICATION USAGE**

48821 Applications using these functions should call *va_end*(*ap*) afterwards to clean up.

48822 **RATIONALE**

48823 None.

48824 **FUTURE DIRECTIONS**

48825 None.

48826 **SEE ALSO**

48827 *fscanf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdarg.h>**, **<stdio.h>**

48828 **CHANGE HISTORY**

48829 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

48830 **NAME**

48831 vfwprintf, vswprintf, vwprintf — wide-character formatted output of a stdarg argument list

48832 **SYNOPSIS**

48833 ```
#include <stdarg.h>
```
48834 ```
#include <stdio.h>
```
48835 ```
#include <wchar.h>
```

48836 ```
int vfwprintf(FILE *restrict stream, const wchar_t *restrict format,
```
48837 ```
    va_list arg);
```
48838 ```
int vswprintf(wchar_t *restrict ws, size_t n,
```
48839 ```
    const wchar_t *restrict format, va_list arg);
```
48840 ```
int vwprintf(const wchar_t *restrict format, va_list arg);
```

48841 **DESCRIPTION**

48842 CX The functionality described on this reference page is aligned with the ISO C standard. Any
48843 conflict between the requirements described here and the ISO C standard is unintentional. This
48844 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

48845 The *vfwprintf*( ), *vswprintf*( ), and *vwprintf*( ) functions shall be equivalent to *fwprintf*( ), *swprintf*( ),
48846 and *wprintf*( ) respectively, except that instead of being called with a variable number of
48847 arguments, they are called with an argument list as defined by **<stdarg.h>**.

48848 These functions shall not invoke the *va_end* macro. However, as these functions do invoke the |
48849 *va_arg* macro, the value of *ap* after the return is unspecified. |

48850 **RETURN VALUE**

48851 Refer to *fwprintf*( ).

48852 **ERRORS**

48853 Refer to *fwprintf*( ).

48854 **EXAMPLES**

48855 None.

48856 **APPLICATION USAGE**

48857 Applications using these functions should call *va_end*(*ap*) afterwards to clean up.

48858 **RATIONALE**

48859 None.

48860 **FUTURE DIRECTIONS**

48861 None.

48862 **SEE ALSO**

48863 *fwprintf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdarg.h>**, **<stdio.h>**,
48864 **<wchar.h>**

48865 **CHANGE HISTORY**

48866 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
48867 (E).

48868 **Issue 6**

48869 The *vfwprintf*( ), *vswprintf*( ), and *vwprintf*( ) prototypes are updated for alignment with the
48870 ISO/IEC 9899:1999 standard. ( )

NAME

vfwscanf, vswscanf, vwscanf — wide-character formattted input of a stdarg list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwscanf(FILE *restrict stream, const wchar_t *restrict format,
    va_list arg);
int vswscanf(const wchar_t *restrict ws, const wchar_t *restrict format,
    va_list arg);
int vwscanf(const wchar_t *restrict format, va_list arg);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *vfwscanf*(), *vswscanf*(), and *vwscanf*() functions shall be equivalent to the *fwscanf*(), *swscanf*(), and *wscanf*() functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined in the **<stdarg.h>** header. These functions shall not invoke the *va_end* macro. As these functions invoke the *va_arg* macro, the value of *ap* after the return is unspecified.

RETURN VALUE

Refer to *fwscanf*().

ERRORS

Refer to *fwscanf*().

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions should call *va_end*(*ap*) afterwards to clean up.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fwscanf(), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdarg.h>**, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

48908 **NAME**

48909 vprintf — format output of a stdarg argument list

48910 **SYNOPSIS**

48911 `#include <stdarg.h>`

48912 `#include <stdio.h>`

48913 `int vprintf(const char *restrict format, va_list ap);` |

48914 **DESCRIPTION** |

48915 Refer to *vfprintf*( ).

48916 **NAME**

48917 vscanf — format input of a stdarg list

48918 **SYNOPSIS**

48919 `#include <stdarg.h>`

48920 `#include <stdio.h>`

48921 `int vscanf(const char *restrict format, va_list arg);`

48922 **DESCRIPTION**

48923 Refer to *vfscanf*( ).

48924 **NAME**

48925 vsnprintf, vsprintf — format output of a stdarg argument list

48926 **SYNOPSIS**

48927 `#include <stdarg.h>`
48928 `#include <stdio.h>`

48929 `int vsnprintf(char *restrict s, size_t n,`
48930 `const char *restrict format, va_list ap);`
48931 `int vsprintf(char *restrict s, const char *restrict format,`
48932 `va_list ap);`

48933 **DESCRIPTION**

48934 Refer to *vfprintf*( ).

48935 **NAME**

48936 vsscanf — format input of a stdarg list

48937 **SYNOPSIS**

48938 `#include <stdarg.h>`
48939 `#include <stdio.h>`

48940 `int vsscanf(const char *restrict s, const char *restrict format,`
48941 `va_list arg);`

48942 **DESCRIPTION**

48943 Refer to *vfscanf*( ).

48944 **NAME**

48945 vswprintf — wide-character formatted output of a stdarg argument list

48946 **SYNOPSIS**

48947 ```
#include <stdarg.h>
```
48948 ```
#include <stdio.h>
```
48949 ```
#include <wchar.h>
```

48950 ```
int vswprintf(wchar_t *restrict ws, size_t n,
```
48951 ```
    const wchar_t *restrict format, va_list arg);
```

48952 **DESCRIPTION**

48953 Refer to *vfwprintf*( ).
```

48954 **NAME**

48955 vswscanf — wide-character formattted input of a stdarg list

48956 **SYNOPSIS**

48957 ```
#include <stdarg.h>
```
48958 ```
#include <stdio.h>
```
48959 ```
#include <wchar.h>
```

48960 ```
int vswscanf(const wchar_t *restrict ws, const wchar_t *restrict format,
```
48961 ```
    va_list arg);
```

48962 **DESCRIPTION**

48963 Refer to *vfwscanf*( ).
```

48964 **NAME**

48965 vwprintf — wide-character formatted output of a stdarg argument list

48966 **SYNOPSIS**

48967 ```
#include <stdarg.h>
```

48968 ```
#include <stdio.h>
```

48969 ```
#include <wchar.h>
```

48970 ```
int vwprintf(const wchar_t *restrict format, va_list arg);
```

48971 **DESCRIPTION**

48972 Refer to *vfwprintf*( ).

48973 **NAME**

48974 vwscanf — wide-character formattted input of a stdarg list

48975 **SYNOPSIS**

48976 `#include <stdarg.h>`
48977 `#include <stdio.h>`
48978 `#include <wchar.h>`

48979 `int vwscanf(const wchar_t *restrict format, va_list arg);`

48980 **DESCRIPTION**

48981 Refer to *vfwscanf*( ).

48982 **NAME**

48983 wait, waitpid — wait for a child process to stop or terminate

48984 **SYNOPSIS**

48985 `#include <sys/wait.h>`

48986 `pid_t wait(int *stat_loc);`
48987 `pid_t waitpid(pid_t pid, int *stat_loc, int options);`

48988 **DESCRIPTION**

48989 The *wait*( ) and *waitpid*( ) functions shall obtain status information pertaining to one of the
48990 caller's child processes. Various options permit status information to be obtained for child
48991 processes that have terminated or stopped. If status information is available for two or more
48992 child processes, the order in which their status is reported is unspecified.

48993 The *wait*( ) function shall suspend execution of the calling thread until status information for one
48994 of the terminated child processes of the calling process is available, or until delivery of a signal
48995 whose action is either to execute a signal-catching function or to terminate the process. If more
48996 than one thread is suspended in *wait*( ) or *waitpid*( ) awaiting termination of the same process,
48997 exactly one thread shall return the process status at the time of the target process termination. If
48998 status information is available prior to the call to *wait*( ), return shall be immediate.

48999 The *waitpid*( ) function shall be equivalent to *wait*( ) if the *pid* argument is (**pid_t**)–1 and the
49000 *options* argument is 0. Otherwise, its behavior shall be modified by the values of the *pid* and
49001 *options* arguments.

49002 The *pid* argument specifies a set of child processes for which *status* is requested. The *waitpid*( )
49003 function shall only return the status of a child process from this set:

49004 • If *pid* is equal to (**pid_t**)–1, *status* is requested for any child process. In this respect, *waitpid*( )
49005 is then equivalent to *wait*( ).

49006 • If *pid* is greater than 0, it specifies the process ID of a single child process for which *status* is
49007 requested.

49008 • If *pid* is 0, *status* is requested for any child process whose process group ID is equal to that of
49009 the calling process.

49010 • If *pid* is less than (**pid_t**)–1, *status* is requested for any child process whose process group ID
49011 is equal to the absolute value of *pid*.

49012 The *options* argument is constructed from the bitwise-inclusive OR of zero or more of the
49013 following flags, defined in the **<sys/wait.h>** header:

49014 XSI WCONTINUED The *waitpid*( ) function shall report the status of any continued child process
49015 specified by *pid* whose status has not been reported since it continued from a
49016 job control stop.

49017 WNOHANG The *waitpid*( ) function shall not suspend execution of the calling thread if
49018 *status* is not immediately available for one of the child processes specified by
49019 *pid*.

49020 WUNTRACED The status of any child processes specified by *pid* that are stopped, and whose
49021 status has not yet been reported since they stopped, shall also be reported to
49022 the requesting process.

49023 XSI If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN, and the
49024 process has no unwaited-for children that were transformed into zombie processes, the calling
49025 thread shall block until all of the children of the process containing the calling thread terminate,
49026 and *wait*( ) and *waitpid*( ) shall fail and set *errno* to [ECHILD].

If *wait*( ) or *waitpid*( ) return because the status of a child process is available, these functions shall return a value equal to the process ID of the child process. In this case, if the value of the argument *stat_loc* is not a null pointer, information shall be stored in the location pointed to by *stat_loc*. The value stored at the location pointed to by *stat_loc* shall be 0 if and only if the status returned is from a terminated child process that terminated by one of the following means:

1. The process returned 0 from *main*( ).
2. The process called *_exit*( ) or *exit*( ) with a *status* argument of 0.
3. The process was terminated because the last thread in the process terminated.

Regardless of its value, this information may be interpreted using the following macros, which are defined in **<sys/wait.h>** and evaluate to integral expressions; the *stat_val* argument is the integer value pointed to by *stat_loc*.

WIFEXITED(*stat_val*)
: Evaluates to a non-zero value if *status* was returned for a child process that terminated normally.

WEXITSTATUS(*stat_val*)
: If the value of WIFEXITED(*stat_val*) is non-zero, this macro evaluates to the low-order 8 bits of the *status* argument that the child process passed to *_exit*( ) or *exit*( ), or the value the child process returned from *main*( ).

WIFSIGNALED(*stat_val*)
: Evaluates to non-zero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught (see **<signal.h>**).

WTERMSIG(*stat_val*)
: If the value of WIFSIGNALED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED(*stat_val*)
: Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

WSTOPSIG(*stat_val*)
: If the value of WIFSTOPPED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

XSI WIFCONTINUED(*stat_val*)
: Evaluates to a non-zero value if status was returned for a child process that has continued from a job control stop.

SPN It is unspecified whether the *status* value returned by calls to *wait*( ) or *waitpid*( ) for processes created by *posix_spawn*( ) or *posix_spawnp*( ) can indicate a WIFSTOPPED(*stat_val*) before subsequent calls to *wait*( ) or *waitpid*( ) indicate WIFEXITED(*stat_val*) as the result of an error detected before the new process image starts executing.

It is unspecified whether the *status* value returned by calls to *wait*( ) or *waitpid*( ) for processes created by *posix_spawn*( ) or *posix_spawnp*( ) can indicate a WIFSIGNALED(*stat_val*) if a signal is sent to the parent's process group after *posix_spawn*( ) or *posix_spawnp*( ) is called.

If the information pointed to by *stat_loc* was stored by a call to *waitpid*( ) that specified the
XSI WUNTRACED flag and did not specify the WCONTINUED flag, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and WIFSTOPPED(**stat_loc*) shall evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to *waitpid*() that specified the XSI WUNTRACED and WCONTINUED flags, exactly one of the macros WIFEXITED(**stat_loc*), XSI WIFSIGNALED(**stat_loc*), WIFSTOPPED(**stat_loc*), and WIFCONTINUED(**stat_loc*) shall evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to *waitpid*() that did not specify the XSI WUNTRACED or WCONTINUED flags, or by a call to the *wait*() function, exactly one of the macros WIFEXITED(**stat_loc*) and WIFSIGNALED(**stat_loc*) shall evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to *waitpid*() that did not specify the XSI WUNTRACED flag and specified the WCONTINUED flag, or by a call to the *wait*() function, XSI exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and WIFCONTINUED(**stat_loc*) shall evaluate to a non-zero value.

If _POSIX_REALTIME_SIGNALS is defined, and the implementation queues the SIGCHLD signal, then if *wait*() or *waitpid*() returns because the status of a child process is available, any pending SIGCHLD signal associated with the process ID of the child process shall be discarded. Any other pending SIGCHLD signals shall remain pending.

Otherwise, if SIGCHLD is blocked, if *wait*() or *waitpid*() return because the status of a child process is available, any pending SIGCHLD signal shall be cleared unless the status of another child process is available.

For all other conditions, it is unspecified whether child *status* will be available when a SIGCHLD signal is delivered.

There may be additional implementation-defined circumstances under which *wait*() or *waitpid*() report *status*. This shall not occur unless the calling process or one of its child processes explicitly makes use of a non-standard extension. In these cases the interpretation of the reported *status* is implementation-defined.

XSI If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes shall be assigned a new parent process ID corresponding to an implementation-defined system process.

RETURN VALUE

If *wait*() or *waitpid*() returns because the status of a child process is available, these functions shall return a value equal to the process ID of the child process for which *status* is reported. If *wait*() or *waitpid*() returns due to the delivery of a signal to the calling process, –1 shall be returned and *errno* set to [EINTR]. If *waitpid*() was invoked with WNOHANG set in *options*, it has at least one child process specified by *pid* for which *status* is not available, and *status* is not available for any process specified by *pid*, 0 is returned. Otherwise, (**pid_t**)–1 shall be returned, and *errno* set to indicate the error.

ERRORS

The *wait*() function shall fail if:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EINTR] The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

The *waitpid*() function shall fail if:

[ECHILD] The process specified by *pid* does not exist or is not a child of the calling process, or the process group specified by *pid* does not exist or does not have any member process that is a child of the calling process.

[EINTR] The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

[EINVAL] The *options* argument is not valid.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

A call to the *wait*( ) or *waitpid*( ) function only returns *status* on an immediate child process of the calling process; that is, a child that was produced by a single *fork*( ) call (perhaps followed by an *exec* or other function calls) from the parent. If a child produces grandchildren by further use of *fork*( ), none of those grandchildren nor any of their descendants affect the behavior of a *wait*( ) from the original parent process. Nothing in this volume of IEEE Std 1003.1-200x prevents an implementation from providing extensions that permit a process to get *status* from a grandchild or any other process, but a process that does not use such extensions must be guaranteed to see *status* from only its direct children.

The *waitpid*( ) function is provided for three reasons:

1. To support job control
2. To permit a non-blocking version of the *wait*( ) function
3. To permit a library routine, such as *system*( ) or *pclose*( ), to wait for its children without interfering with other terminated children for which the process has not waited

The first two of these facilities are based on the *wait3*( ) function provided by 4.3 BSD. The function uses the *options* argument, which is equivalent to an argument to *wait3*( ). The WUNTRACED flag is used only in conjunction with job control on systems supporting job control. Its name comes from 4.3 BSD and refers to the fact that there are two types of stopped processes in that implementation: processes being traced via the *ptrace*( ) debugging facility and (untraced) processes stopped by job control signals. Since *ptrace*( ) is not part of this volume of IEEE Std 1003.1-200x, only the second type is relevant. The name WUNTRACED was retained because its usage is the same, even though the name is not intuitively meaningful in this context.

The third reason for the *waitpid*( ) function is to permit independent sections of a process to spawn and wait for children without interfering with each other. For example, the following problem occurs in developing a portable shell, or command interpreter:

```
stream = popen("/bin/true");
(void) system("sleep 100");
(void) pclose(stream);
```

On all historical implementations, the final *pclose*( ) fails to reap the *wait*( ) *status* of the *popen*( ).

The status values are retrieved by macros, rather than given as specific bit encodings as they are in most historical implementations (and thus expected by existing programs). This was necessary to eliminate a limitation on the number of signals an implementation can support that was inherent in the traditional encodings. This volume of IEEE Std 1003.1-200x does require that a *status* value of zero corresponds to a process calling *_exit*(0), as this is the most common encoding expected by existing programs. Some of the macro names were adopted from 4.3 BSD.

These macros syntactically operate on an arbitrary integer value. The behavior is undefined unless that value is one stored by a successful call to *wait*( ) or *waitpid*( ) in the location pointed to by the *stat_loc* argument. An early proposal attempted to make this clearer by specifying each

argument as **stat_loc* rather than *stat_val*. However, that did not follow the conventions of other specifications in this volume of IEEE Std 1003.1-200x or traditional usage. It also could have implied that the argument to the macro must literally be **stat_loc*; in fact, that value can be stored or passed as an argument to other functions before being interpreted by these macros.

The extension that affects *wait*( ) and *waitpid*( ) and is common in historical implementations is the *ptrace*( ) function. It is called by a child process and causes that child to stop and return a *status* that appears identical to the *status* indicated by WIFSTOPPED. The *status* of *ptrace*( ) children is traditionally returned regardless of the WUNTRACED flag (or by the *wait*( ) function). Most applications do not need to concern themselves with such extensions because they have control over what extensions they or their children use. However, applications, such as command interpreters, that invoke arbitrary processes may see this behavior when those arbitrary processes misuse such extensions.

Implementations that support *core* file creation or other implementation-defined actions on termination of some processes traditionally provide a bit in the *status* returned by *wait*( ) to indicate that such actions have occurred.

Allowing the *wait*( ) family of functions to discard a pending SIGCHLD signal that is associated with a successfully waited-for child process puts them into the *sigwait*( ) and *sigwaitinfo*( ) category with respect to SIGCHLD.

This definition allows implementations to treat a pending SIGCHLD signal as accepted by the process in *wait*( ), with the same meaning of ‘‘accepted’’ as when that word is applied to the *sigwait*( ) family of functions.

Allowing the *wait*( ) family of functions to behave this way permits an implementation to be able to deal precisely with SIGCHLD signals.

In particular, an implementation that does accept (discard) the SIGCHLD signal can make the following guarantees regardless of the queuing depth of signals in general (the list of waitable children can hold the SIGCHLD queue):

1. If a SIGCHLD signal handler is established via *sigaction*( ) without the SA_RESETHAND flag, SIGCHLD signals can be accurately counted; that is, exactly one SIGCHLD signal will be delivered to or accepted by the process for every child process that terminates.
2. A single *wait*( ) issued from a SIGCHLD signal handler can be guaranteed to return immediately with status information for a child process.
3. When SA_SIGINFO is requested, the SIGCHLD signal handler can be guaranteed to receive a non-NULL pointer to a **siginfo_t** structure that describes a child process for which a wait via *waitpid*( ) or *waitid*( ) will not block or fail.
4. The *system*( ) function will not cause a processs SIGCHLD handler to be called as a result of the *fork*( )/*exec* executed within *system*( ) because *system*( ) will accept the SIGCHLD signal when it performs a *waitpid*( ) for its child process. This is a desirable behavior of *system*( ) so that it can be used in a library without causing side effects to the application linked with the library.

An implementation that does not permit the *wait*( ) family of functions to accept (discard) a pending SIGCHLD signal associated with a successfully waited-for child, cannot make the guarantees described above for the following reasons:

Guarantee #1

Although it might be assumed that reliable queuing of all SIGCHLD signals generated by the system can make this guarantee, the counter example is the case of a process that blocks SIGCHLD and performs an indefinite loop of *fork*( )/*wait*( ) operations. If the

implementation supports queued signals, then eventually the system will run out of memory for the queue. The guarantee cannot be made because there must be some limit to the depth of queuing.

Guarantees #2 and #3

These cannot be guaranteed unless the *wait*() family of functions accepts the SIGCHLD signal. Otherwise, a *fork*()/*wait*() executed while SIGCHLD is blocked (as in the *system*() function) will result in an invocation of the handler when SIGCHLD is unblocked, after the process has disappeared.

Guarantee #4

Although possible to make this guarantee, *system*() would have to set the SIGCHLD handler to SIG_DFL so that the SIGCHLD signal generated by its *fork*() would be discarded (the SIGCHLD default action is to be ignored), then restore it to its previous setting. This would have the undesirable side effect of discarding all SIGCHLD signals pending to the process.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit*(), *fork*(), *waitid*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<sys/wait.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The processing of the SIGCHLD signal and the [ECHILD] error is clarified.

The semantics of WIFSTOPPED(*stat_val*), WIFEXITED(*stat_val*), and WIFSIGNALED(*stat_val*) are defined with respect to *posix_spawn*() or *posix_spawnp*() for alignment with IEEE Std 1003.1d-1999.

The DESCRIPTION is updated for alignment with the ISO/IEC 9899: 1999 standard.

NAME

waitid — wait for a child process to change state

SYNOPSIS

XSI
```
#include <sys/wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

DESCRIPTION

The *waitid*( ) function shall suspend the calling thread until one child of the process containing the calling thread changes state. It records the current state of a child in the structure pointed to by *infop*. If a child process changed state prior to the call to *waitid*( ), *waitid*( ) shall return immediately. If more than one thread is suspended in *wait*( ) or *waitpid*( ) waiting termination of the same process, exactly one thread shall return the process status at the time of the target process termination.

The *idtype* and *id* arguments are used to specify which children *waitid*( ) waits for.

If *idtype* is P_PID, *waitid*( ) shall wait for the child with a process ID equal to (**pid_t**)*id*.

If *idtype* is P_PGID, *waitid*( ) shall wait for any child with a process group ID equal to (**pid_t**)*id*.

If *idtype* is P_ALL, *waitid*( ) shall wait for any children and *id* is ignored.

The *options* argument is used to specify which state changes *waitid*( ) shall wait for. It is formed by OR'ing together one or more of the following flags:

WEXITED	Wait for processes that have exited.
WSTOPPED	Status shall be returned for any child that has stopped upon receipt of a signal.
WCONTINUED	Status shall be returned for any child that was stopped and has been continued.
WNOHANG	Return immediately if there are no children to wait for.
WNOWAIT	Keep the process whose status is returned in *infop* in a waitable state. This shall not affect the state of the process; the process may be waited for again after this call completes.

The application shall ensure that the *infop* argument points to a **siginfo_t** structure. If *waitid*( ) returns because a child process was found that satisfied the conditions indicated by the arguments *idtype* and *options*, then the structure pointed to by *infop* shall be filled in by the system with the status of the process. The *si_signo* member shall always be equal to SIGCHLD.

RETURN VALUE

If WNOHANG was specified and there are no children to wait for, 0 shall be returned. If *waitid*( ) returns due to the change of state of one of its children, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *waitid*( ) function shall fail if:

[ECHILD]	The calling process has no existing unwaited-for child processes.
[EINTR]	The *waitid*( ) function was interrupted by a signal.
[EINVAL]	An invalid value was specified for *options*, or *idtype* and *id* specify an invalid set of processes.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit*( ), *wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/wait.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

49300 **NAME**

49301 waitpid — wait for a child process to stop or terminate

49302 **SYNOPSIS**

49303 `#include <sys/wait.h>`

49304 `pid_t waitpid(pid_t pid, int *stat_loc, int options);`

49305 **DESCRIPTION**

49306 Refer to *wait*( ).

NAME

wcrtomb — convert a wide-character code to a character (restartable)

SYNOPSIS

```
#include <stdio.h>

size_t wcrtomb(char *restrict s, wchar_t wc, mbstate_t *restrict ps);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

If *s* is a null pointer, the *wcrtomb*( ) function shall be equivalent to the call:

```
wcrtomb(buf, L'\0', ps)
```

where *buf* is an internal buffer.

If *s* is not a null pointer, the *wcrtomb*( ) function shall determine the number of bytes needed to represent the character that corresponds to the wide character given by *wc* (including any shift sequences), and store the resulting bytes in the array whose first element is pointed to by *s*. At most {MB_CUR_MAX} bytes are stored. If *wc* is a null wide character, a null byte shall be stored, preceded by any shift sequence needed to restore the initial shift state. The resulting state described shall be the initial conversion state.

If *ps* is a null pointer, the *wcrtomb*( ) function shall use its own internal **mbstate_t** object, which is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* shall be used to completely describe the current conversion state of the associated character sequence. The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *wcrtomb*( ).

CX If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS functions, the application shall ensure that the *wcrtomb*( ) function is called with a non-NULL *ps* argument.

The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

The *wcrtomb*( ) function shall return the number of bytes stored in the array object (including any shift sequences). When *wc* is not a valid wide character, an encoding error shall occur. In this case, the function shall store the value of the macros [EILSEQ] in *errno* and shall return (**size_t**)–1; the conversion state shall be undefined.

ERRORS

The *wcrtomb*( ) function may fail if:

CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

[EILSEQ] Invalid wide-character code is detected.

49343 **EXAMPLES**

49344 None.

49345 **APPLICATION USAGE**

49346 None.

49347 **RATIONALE**

49348 None.

49349 **FUTURE DIRECTIONS**

49350 None.

49351 **SEE ALSO**

49352 *mbsinit*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

49353 **CHANGE HISTORY**

49354 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
49355 (E).

49356 **Issue 6**

49357 In the DESCRIPTION, a note on using this function in a threaded application is added.

49358 Extensions beyond the ISO C standard are now marked.

49359 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

49360 The *wcrtomb*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcscat — concatenate two wide-character strings

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcscat(wchar_t *restrict ws1, const wchar_t *restrict ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcscat*( ) function shall append a copy of the wide-character string pointed to by *ws2* (including the terminating null wide-character code) to the end of the wide-character string pointed to by *ws1*. The initial wide-character code of *ws2* shall overwrite the null wide-character code at the end of *ws1*. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *wcscat*( ) function shall return *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsncat( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The Open Group Corrigendum U040/2 is applied. In the RETURN VALUE section, *s1* is changed to *ws1*.

The *wcscat*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcschr — wide-character string scanning operation

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcschr(const wchar_t *ws, wchar_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcschr*( ) function shall locate the first occurrence of *wc* in the wide-character string pointed to by *ws*. The application shall ensure that the value of *wc* is a character representable as a type **wchar_t** and a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code is considered to be part of the wide-character string.

RETURN VALUE

Upon completion, *wcschr*( ) shall return a pointer to the wide-character code, or a null pointer if the wide-character code is not found.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsrchr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

wcscmp — compare two wide-character strings

SYNOPSIS

```
#include <wchar.h>

int wcscmp(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcscmp*( ) function shall compare the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*.

The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared.

RETURN VALUE

Upon completion, *wcscmp*( ) shall return an integer greater than, equal to, or less than 0, if the wide-character string pointed to by *ws1* is greater than, equal to, or less than the wide-character string pointed to by *ws2*, respectively.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsncmp( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

NAME

wcscoll — wide-character string comparison using collating information

SYNOPSIS

```
#include <wchar.h>

int wcscoll(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcscoll*( ) function shall compare the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*, both interpreted as appropriate to the *LC_COLLATE* category of the current locale.

CX The *wcscoll*( ) function shall not change the setting of *errno* if successful.

An application wishing to check for error situations should set *errno* to 0 before calling *wcscoll*( ). If *errno* is non-zero on return, an error has occurred.

RETURN VALUE

Upon successful completion, *wcscoll*( ) shall return an integer greater than, equal to, or less than 0, according to whether the wide-character string pointed to by *ws1* is greater than, equal to, or less than the wide-character string pointed to by *ws2*, when both are interpreted as appropriate CX to the current locale. On error, *wcscoll*( ) shall set *errno*, but no return value is reserved to indicate an error.

ERRORS

The *wcscoll*( ) function may fail if:

CX [EINVAL] The *ws1* or *ws2* arguments contain wide-character codes outside the domain of the collating sequence.

EXAMPLES

None.

APPLICATION USAGE

The *wcsxfrm*( ) and *wcscmp*( ) functions should be used for sorting large lists.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscmp( ), *wcsxfrm*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

NAME

wcscpy — copy a wide-character string

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcscpy(wchar_t *restrict ws1, const wchar_t *restrict ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcscpy*( ) function shall copy the wide-character string pointed to by *ws2* (including the terminating null wide-character code) into the array pointed to by *ws1*. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *wcscpy*( ) function shall return *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsncpy( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The *wcscpy*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

wcscspn — get length of a complementary wide substring

SYNOPSIS

```
#include <wchar.h>

size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcscspn*( ) function shall compute the length (in wide characters) of the maximum initial segment of the wide-character string pointed to by *ws1* which consists entirely of wide-character codes *not* from the wide-character string pointed to by *ws2*.

RETURN VALUE

The *wcscspn*( ) function shall return the length of the initial substring of *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsspn( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The RETURN VALUE section is updated to indicate that *wcscspn*( ) returns the length of *ws1*, rather than *ws1* itself.

NAME

wcsftime — convert date and time to a wide-character string

SYNOPSIS

```
#include <wchar.h>

size_t wcsftime(wchar_t *restrict wcs, size_t maxsize,
    const wchar_t *restrict format, const struct tm *restrict timeptr);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsftime*( ) function shall be equivalent to the *strftime*( ) function, except that:

- The argument *wcs* points to the initial element of an array of wide characters into which the generated output is to be placed.
- The argument *maxsize* indicates the maximum number of wide characters to be placed in the output array.
- The argument *format* is a wide-character string and the conversion specifications are replaced by corresponding sequences of wide characters.
- The return value indicates the number of wide characters placed in the output array.

If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

If the total number of resulting wide-character codes including the terminating null wide-character code is no more than *maxsize*, *wcsftime*( ) shall return the number of wide-character codes placed into the array pointed to by *wcs*, not including the terminating null wide-character code. Otherwise, zero is returned and the contents of the array are unspecified.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strftime( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 5

Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of the *format* argument is changed from **const char *** to **const wchar_t ***.

49602 **Issue 6**

49603 The *wcsftime*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

wcslen — get wide-character string length

SYNOPSIS

```
#include <wchar.h>

size_t wcslen(const wchar_t *ws);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcslen*() function shall compute the number of wide-character codes in the wide-character string to which *ws* points, not including the terminating null wide-character code.

RETURN VALUE

The *wcslen*() function shall return the length of *ws*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

NAME

wcsncat — concatenate a wide-character string with part of another

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsncat(wchar_t *restrict ws1, const wchar_t *restrict ws2,
    size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsncat*( ) function shall append not more than *n* wide-character codes (a null wide-character code and wide-character codes that follow it are not appended) from the array pointed to by *ws2* to the end of the wide-character string pointed to by *ws1*. The initial wide-character code of *ws2* shall overwrite the null wide-character code at the end of *ws1*. A terminating null wide-character code shall always be appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *wcsncat*( ) function shall return *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscat( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The *wcsncat*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

wcsncmp — compare part of two wide-character strings

SYNOPSIS

```
#include <wchar.h>

int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsncmp*( ) function shall compare not more than *n* wide-character codes (wide-character codes that follow a null wide-character code are not compared) from the array pointed to by *ws1* to the array pointed to by *ws2*.

The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared.

RETURN VALUE

Upon successful completion, *wcsncmp*( ) shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by *ws1* is greater than, equal to, or less than the possibly null-terminated array pointed to by *ws2*, respectively.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscmp( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

NAME

wcsncpy — copy part of a wide-character string

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsncpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,
    size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsncpy*( ) function shall copy not more than *n* wide-character codes (wide-character codes that follow a null wide-character code are not copied) from the array pointed to by *ws2* to the array pointed to by *ws1*. If copying takes place between objects that overlap, the behavior is undefined.

If the array pointed to by *ws2* is a wide-character string that is shorter than *n* wide-character codes, null wide-character codes shall be appended to the copy in the array pointed to by *ws1*, until *n* wide-character codes in all are written.

RETURN VALUE

The *wcsncpy*( ) function shall return *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

If there is no null wide-character code in the first *n* wide-character codes of the array pointed to by *ws2*, the result is not null-terminated.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscpy( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The *wcsncpy*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

wcspbrk — scan wide-character string for a wide-character code

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcspbrk*( ) function shall locate the first occurrence in the wide-character string pointed to by *ws1* of any wide-character code from the wide-character string pointed to by *ws2*.

RETURN VALUE

Upon successful completion, *wcspbrk*( ) shall return a pointer to the wide-character code or a null pointer if no wide-character code from *ws2* occurs in *ws1*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcschr( ), *wcsrchr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

NAME

wcsrchr — wide-character string scanning operation

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsrchr*( ) function shall locate the last occurrence of *wc* in the wide-character string pointed to by *ws*. The application shall ensure that the value of *wc* is a character representable as a type **wchar_t** and a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code shall be considered to be part of the wide-character string.

RETURN VALUE

Upon successful completion, *wcsrchr*( ) shall return a pointer to the wide-character code or a null pointer if *wc* does not occur in the wide-character string.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcschr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

wcsrtombs — convert a wide-character string to a character string (restartable)

SYNOPSIS

```
#include <wchar.h>

size_t wcsrtombs(char *restrict dst, const wchar_t **restrict src,
    size_t len, mbstate_t *restrict ps);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsrtombs*( ) function shall convert a sequence of wide characters from the array indirectly pointed to by *src* into a sequence of corresponding characters, beginning in the conversion state described by the object pointed to by *ps*. If *dst* is not a null pointer, the converted characters shall then be stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null wide character, which shall also be stored. Conversion shall stop earlier in the following cases:

- When a code is reached that does not correspond to a valid character
- When the next character would exceed the limit of *len* total bytes to be stored in the array pointed to by *dst* (and *dst* is not a null pointer)

Each conversion shall take place as if by a call to the *wcrtomb*( ) function.

If *dst* is not a null pointer, the pointer object pointed to by *src* shall be assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described shall be the initial conversion state.

If *ps* is a null pointer, the *wcsrtombs*( ) function shall use its own internal **mbstate_t** object, which is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* shall be used to completely describe the current conversion state of the associated character sequence. The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *wcsrtombs*( ).

CX If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS functions, the application shall ensure that the *wcsrtombs*( ) function is called with a non-NULL *ps* argument.

The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

If conversion stops because a code is reached that does not correspond to a valid character, an encoding error occurs. In this case, the *wcsrtombs*( ) function shall store the value of the macro [EILSEQ] in *errno* and return (**size_t**)–1; the conversion state is undefined. Otherwise, it shall return the number of bytes in the resulting character sequence, not including the terminating null (if any).

ERRORS

The *wcsrtombs*( ) function may fail if:

CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

[EILSEQ] A wide-character code does not correspond to a valid character.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbsinit( ), *wcrtomb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

In the DESCRIPTION, a note on using this function in a threaded application is added.

Extensions beyond the ISO C standard are now marked.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *wcsrtombs*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcsspn — get length of a wide substring

SYNOPSIS

```
#include <wchar.h>

size_t wcsspn(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsspn*( ) function shall compute the length (in wide characters) of the maximum initial segment of the wide-character string pointed to by *ws1* which consists entirely of wide-character codes from the wide-character string pointed to by *ws2*.

RETURN VALUE

The *wcsspn*( ) function shall return the length of the initial substring of *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscspn( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The RETURN VALUE section is updated to indicate that *wcsspn*( ) returns the length of *ws1* rather that *ws1* itself.

NAME

wcsstr — find a wide-character substring

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsstr(const wchar_t *restrict ws1, const wchar_t *restrict ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsstr*( ) function shall locate the first occurrence in the wide-character string pointed to by *ws1* of the sequence of wide characters (excluding the terminating null wide character) in the wide-character string pointed to by *ws2*.

RETURN VALUE

Upon successful completion, *wcsstr*( ) shall return a pointer to the located wide-character string, or a null pointer if the wide-character string is not found.

If *ws2* points to a wide-character string with zero length, the function shall return *ws1*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcschr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The *wcsstr*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcstod, wcstof, wcstold — convert a wide-character string to a double-precision number

SYNOPSIS

```
#include <wchar.h>

double wcstod(const wchar_t *restrict nptr, wchar_t **restrict endptr);
float wcstof(const wchar_t *restrict nptr, wchar_t **restrict endptr);
long double wcstold(const wchar_t *restrict nptr,
    wchar_t **restrict endptr);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall convert the initial portion of the wide-character string pointed to by *nptr* to **double**, **float**, and **long double** representation, respectively. First, they shall decompose the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by *iswspace*( ))
2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
3. A final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide-character code of the input wide-character string

Then they shall attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent part
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character, then an optional binary exponent part
- One of INF or INFINITY, or any other wide string equivalent except for case
- One of NAN or NAN(*n-wchar-sequence*$_{opt}$), or any other wide string ignoring case in the NAN part, where:

```
n-wchar-sequence:
    digit
    nondigit
    n-wchar-sequence digit
    n-wchar-sequence nondigit
```

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the radix character (whichever occurs first) shall be interpreted as a floating constant according to the rules of the C language, except that the radix character shall be used in place of a period, and that if neither an exponent part nor a radix character appears in a decimal floating-point number, or if a binary exponent part does not

appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero shall be assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence shall be interpreted as negated. A wide-character sequence INF or INFINITY shall be interpreted as an infinity, if representable in the return type, else as if it were a floating constant that is too large for the range of the return type. A wide-character sequence NAN or NAN(*n-wchar-sequence*$_{opt}$) shall be interpreted as a quiet NaN, if supported in the return type, else as if it were a subject sequence part that does not have the expected form; the meaning of the *n*-wchar sequences is implementation-defined. A pointer to the final wide string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the conversion shall be rounded in an implementation-defined manner.

CX The radix character shall be as defined in the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period (`'.'`).

CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX The *wcstod*( ) function shall not change the setting of *errno* if successful.

Since 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set *errno* to 0, then call *wcstod*( ), *wcstof*( ), or *wcstold*( ), then check *errno*.

RETURN VALUE

Upon successful completion, these functions shall return the converted value. If no conversion could be performed, 0 shall be returned CX and *errno* may be set to [EINVAL].

If the correct value is outside the range of representable values, plus or minus HUGE_VAL, HUGE_VALF, or HUGE_VALL shall be returned (according to the sign of the value), and *errno* shall be set to [ERANGE].

If the correct value would cause underflow, a value whose magnitude is no greater than the smallest normalized positive number in the return type shall be returned and *errno* set to [ERANGE].

ERRORS

The *wcstod*( ) function shall fail if:

[ERANGE] The value to be returned would cause overflow or underflow.

The *wcstod*( ) function may fail if:

CX [EINVAL] No conversion could be performed.

50001 **EXAMPLES**

50002 None.

50003 **APPLICATION USAGE**

50004 If the subject sequence has the hexadecimal form and FLT_RADIX is not a power of 2, and the
50005 result is not exactly representable, the result should be one of the two numbers in the
50006 appropriate internal format that are adjacent to the hexadecimal floating source value, with the
50007 extra stipulation that the error should have a correct sign for the current rounding direction.

50008 If the subject sequence has the decimal form and at most DECIMAL_DIG (defined in **<float.h>**)
50009 significant digits, the result should be correctly rounded. If the subject sequence *D* has the
50010 decimal form and more than DECIMAL_DIG significant digits, consider the two bounding,
50011 adjacent decimal strings *L* and *U*, both having DECIMAL_DIG significant digits, such that the
50012 values of *L*, *D*, and *U* satisfy `"L <= D <= U"`. The result should be one of the (equal or
50013 adjacent) values that would be obtained by correctly rounding *L* and *U* according to the current
50014 rounding direction, with the extra stipulation that the error with respect to *D* should have a
50015 correct sign for the current rounding direction.

50016 **RATIONALE**

50017 None.

50018 **FUTURE DIRECTIONS**

50019 None.

50020 **SEE ALSO**

50021 *iswspace*( ), *localeconv*( ), *scanf*( ), *setlocale*( ), *wcstol*( ), the Base Definitions volume of
50022 IEEE Std 1003.1-200x, **<float.h>**, **<wchar.h>**, the Base Definitions volume of
50023 IEEE Std 1003.1-200x, Chapter 7, Locale

50024 **CHANGE HISTORY**

50025 First released in Issue 4. Derived from the MSE working draft.

50026 **Issue 5**

50027 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

50028 **Issue 6**

50029 Extensions beyond the ISO C standard are now marked.

50030 The following new requirements on POSIX implementations derive from alignment with the
50031 Single UNIX Specification:

50032 - In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
50033 added if no conversion could be performed.

50034 The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

50035 - The *wcstod*( ) prototype is updated.

50036 - The *wcstof*( ) and *wcstold*( ) functions are added.

50037 - If the correct value for *wcstod*( ) would cause underflow, the return value changed from 0 (as
50038 specified in Issue 5) to the smallest normalized positive number.

50039 - The DESCRIPTION, RETURN VALUE, and APPLICATION USAGE sections are extensively
50040 updated.

50041 ISO / IEC 9899: 1999 standard, Technical Corrigendum No. 1 is incorporated.

50042 **NAME**

50043 wcstof — convert a wide-character string to a double-precision number

50044 **SYNOPSIS**

50045 `#include <wchar.h>`

50046 `float wcstof(const wchar_t *restrict nptr, wchar_t **restrict endptr);`

50047 **DESCRIPTION**

50048 Refer to *wcstod*( ).

50049 **NAME**

50050 wcstoimax, wcstoumax — convert wide-character string to integer type

50051 **SYNOPSIS**

50052 ```
#include <stddef.h>
```

50053 ```
#include <inttypes.h>
```

50054 ```
intmax_t wcstoimax(const wchar_t *restrict nptr,
```

50055 ```
    wchar_t **restrict endptr, int base);
```

50056 ```
uintmax_t wcstoumax(const wchar_t *restrict nptr,
```

50057 ```
    wchar_t **restrict endptr, int base);
```

50058 **DESCRIPTION**

50059 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50060 conflict between the requirements described here and the ISO C standard is unintentional. This
50061 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50062 These functions shall be equivalent to the *wcstol*( ), *wcstoll*( ), *wcstoul*( ), and *wcstoull*( ) functions,
50063 respectively, except that the initial portion of the wide string shall be converted to **intmax_t** and
50064 **uintmax_t** representation, respectively.

50065 **RETURN VALUE**

50066 These functions shall return the converted value, if any.

50067 If no conversion could be performed, zero shall be returned. If the correct value is outside the
50068 range of representable values, {INTMAX_MAX}, {INTMAX_MIN}, or {UINTMAX_MAX} shall
50069 be returned (according to the return type and sign of the value, if any), and *errno* shall be set to
50070 [ERANGE].

50071 **ERRORS**

50072 These functions shall fail if:

50073 [EINVAL] The value of *base* is not supported.

50074 [ERANGE] The value to be returned is not representable.

50075 These functions may fail if:

50076 [EINVAL] No conversion could be performed.

50077 **EXAMPLES**

50078 None.

50079 **APPLICATION USAGE**

50080 None.

50081 **RATIONALE**

50082 None.

50083 **FUTURE DIRECTIONS**

50084 None.

50085 **SEE ALSO**

50086 *wcstol*( ), *wcstoul*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<inttypes.h>**,
50087 **<stddef.h>**

50088 **CHANGE HISTORY**

50089 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

50090 **NAME**

50091 wcstok — split wide-character string into tokens

50092 **SYNOPSIS**

50093 ```
#include <wchar.h>
```

50094 ```
wchar_t *wcstok(wchar_t *restrict ws1, const wchar_t *restrict ws2,
50095     wchar_t **restrict ptr);
```

50096 **DESCRIPTION**

50097 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50098 conflict between the requirements described here and the ISO C standard is unintentional. This
50099 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50100 A sequence of calls to *wcstok*( ) shall break the wide-character string pointed to by *ws1* into a |
50101 sequence of tokens, each of which shall be delimited by a wide-character code from the wide- |
50102 character string pointed to by *ws2*. The *ptr* argument points to a caller-provided **wchar_t** pointer |
50103 into which the *wcstok*( ) function shall store information necessary for it to continue scanning the |
50104 same wide-character string. |

50105 The first call in the sequence has *ws1* as its first argument, and is followed by calls with a null
50106 pointer as their first argument. The separator string pointed to by *ws2* may be different from call
50107 to call.

50108 The first call in the sequence shall search the wide-character string pointed to by *ws1* for the first |
50109 wide-character code that is *not* contained in the current separator string pointed to by *ws2*. If no
50110 such wide-character code is found, then there are no tokens in the wide-character string pointed
50111 to by *ws1* and *wcstok*( ) shall return a null pointer. If such a wide-character code is found, it shall |
50112 be the start of the first token. |

50113 The *wcstok*( ) function shall then search from there for a wide-character code that *is* contained in |
50114 the current separator string. If no such wide-character code is found, the current token extends
50115 to the end of the wide-character string pointed to by *ws1*, and subsequent searches for a token |
50116 shall return a null pointer. If such a wide-character code is found, it shall be overwritten by a |
50117 null wide character, which terminates the current token. The *wcstok*( ) function shall save a |
50118 pointer to the following wide-character code, from which the next search for a token shall start. |

50119 Each subsequent call, with a null pointer as the value of the first argument, shall start searching |
50120 from the saved pointer and behave as described above. |

50121 The implementation shall behave as if no function calls *wcstok*( ).

50122 **RETURN VALUE**

50123 Upon successful completion, the *wcstok*( ) function shall return a pointer to the first wide-
50124 character code of a token. Otherwise, if there is no token, *wcstok*( ) shall return a null pointer.

50125 **ERRORS**

50126 No errors are defined.

50127 **EXAMPLES**

50128 None.

50129 **APPLICATION USAGE**

50130 None.

50131 **RATIONALE**

50132 None.

50133 **FUTURE DIRECTIONS**

50134 None.

50135 **SEE ALSO**

50136 The Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

50137 **CHANGE HISTORY**

50138 First released in Issue 4.

50139 **Issue 5**

50140 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, a third argument is
50141 added to the definition of this function in the SYNOPSIS.

50142 **Issue 6**

50143 The *wcstok*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcstol, wcstoll — convert a wide-character string to a long integer

SYNOPSIS

```
#include <wchar.h>

long wcstol(const wchar_t *restrict nptr, wchar_t **restrict endptr,
    int base);
long long wcstoll(const wchar_t *restrict nptr,
    wchar_t **restrict endptr, int base);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall convert the initial portion of the wide-character string pointed to by *nptr* to **long**, **long long**, **unsigned long**, and **unsigned long long** representation, respectively. First, they shall decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by *iswspace*( ))
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide-character code of the input wide-character string

Then they shall attempt to convert the subject sequence to an integer, and return the result.

If *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a `'+'` or `'−'` sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix `'0'` optionally followed by a sequence of the digits `'0'` to `'7'` only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters `'a'` (or `'A'`) to `'f'` (or `'F'`) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a `'+'` or `'−'` sign, but not including an integer suffix. The letters from `'a'` (or `'A'`) to `'z'` (or `'Z'`) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* shall be permitted. If the value of *base* is 16, the wide-character code representations of 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first non-white-space wide-character code that is of the expected form. The subject sequence contains no wide-character codes if the input wide-character string is empty or consists entirely of white-space wide-character code, or if the first non-white-space wide-character code is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and *base* is 0, the sequence of wide-character codes starting with the first digit shall be interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to the final wide-character string shall be stored in the object pointed to by *endptr*, provided that *endptr* is

not a null pointer.

CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX These functions shall not change the setting of *errno* if successful.

Since 0, {LONG_MIN} or {LLONG_MIN} and {LONG_MAX} or {LLONG_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set *errno* to 0, then call *wcstol*( ) or *wcstoll*( ), then check *errno*.

RETURN VALUE

Upon successful completion, these functions shall return the converted value, if any. If no CX conversion could be performed, 0 shall be returned and *errno* may be set to indicate the error. If the correct value is outside the range of representable values, {LONG_MIN}, {LONG_MAX}, {LLONG_MIN}, or {LLONG_MAX} shall be returned (according to the sign of the value), and *errno* set to [ERANGE].

ERRORS

These functions shall fail if:

CX [EINVAL] The value of *base* is not supported.

[ERANGE] The value to be returned is not representable.

These functions may fail if:

CX [EINVAL] No conversion could be performed.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalpha( ), *scanf*( ), *wcstod*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

- The *wcstol*( ) prototype is updated.
- The *wcstoll*( ) function is added.

50235 **NAME**

50236 wcstold — convert a wide-character string to a double-precision number

50237 **SYNOPSIS**

50238
```
#include <wchar.h>
```

50239
```
long double wcstold(const wchar_t *restrict nptr,
```
50240
```
    wchar_t **restrict endptr);
```

50241 **DESCRIPTION**

50242 Refer to *wcstod*( ).

50243 **NAME**

50244 wcstoll — convert a wide-character string to a long integer

50245 **SYNOPSIS**

50246 ```
#include <wchar.h>
```

50247 ```
long long wcstoll(const wchar_t *restrict nptr,
```

50248 ```
    wchar_t **restrict endptr, int base);
```

50249 **DESCRIPTION**

50250 Refer to *wcstol*( ).
```

50251 **NAME**

50252 wcstombs — convert a wide-character string to a character string

50253 **SYNOPSIS**

50254 ```
#include <stdlib.h>
```

50255 ```
size_t wcstombs(char *restrict s, const wchar_t *restrict pwcs,
```
50256 ```
    size_t n);
```

50257 **DESCRIPTION**

50258 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50259 conflict between the requirements described here and the ISO C standard is unintentional. This
50260 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50261 The *wcstombs*( ) function shall convert the sequence of wide-character codes that are in the array
50262 pointed to by *pwcs* into a sequence of characters that begins in the initial shift state and store |
50263 these characters into the array pointed to by *s*, stopping if a character would exceed the limit of *n* |
50264 total bytes or if a null byte is stored. Each wide-character code shall be converted as if by a call to |
50265 *wctomb*( ), except that the shift state of *wctomb*( ) shall not be affected. |

50266 The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

50267 No more than *n* bytes shall be modified in the array pointed to by *s*. If copying takes place
50268 CX between objects that overlap, the behavior is undefined. If *s* is a null pointer, *wcstombs*( ) shall
50269 return the length required to convert the entire array regardless of the value of *n*, but no values
50270 are stored.

50271 The *wcstombs*( ) function need not be reentrant. A function that is not required to be reentrant is
50272 not required to be thread-safe.

50273 **RETURN VALUE**

50274 If a wide-character code is encountered that does not correspond to a valid character (of one or
50275 more bytes each), *wcstombs*( ) shall return (**size_t**)–1. Otherwise, *wcstombs*( ) shall return the
50276 number of bytes stored in the character array, not including any terminating null byte. The array
50277 shall not be null-terminated if the value returned is *n*.

50278 **ERRORS**

50279 The *wcstombs*( ) function may fail if:

50280 CX [EILSEQ] A wide-character code does not correspond to a valid character.

50281 **EXAMPLES**

50282 None.

50283 **APPLICATION USAGE**

50284 None.

50285 **RATIONALE**

50286 None.

50287 **FUTURE DIRECTIONS**

50288 None.

50289 **SEE ALSO**

50290 *mblen*( ), *mbtowc*( ), *mbstowcs*( ), *wctomb*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
50291 **<stdlib.h>**
```

50292 **CHANGE HISTORY**

50293 First released in Issue 4. Derived from the ISO C standard.

50294 **Issue 6**

50295 The following new requirements on POSIX implementations derive from alignment with the
50296 Single UNIX Specification:

50297 • The DESCRIPTION states the effect of when *s* is a null pointer.

50298 • The [EILSEQ] error condition is added.

50299 The *wcstombs*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

wcstoul, wcstoull — convert a wide-character string to an unsigned long

SYNOPSIS

```
#include <wchar.h>

unsigned long wcstoul(const wchar_t *restrict nptr,
    wchar_t **restrict endptr, int base);
unsigned long long wcstoull(const wchar_t *restrict nptr,
    wchar_t **restrict endptr, int base);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcstoul*() and *wcstoull*() functions shall convert the initial portion of the wide-character string pointed to by *nptr* to **unsigned long** and **unsigned long long** representation, respectively. First, they shall decompose the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by *iswspace*())
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide-character code of the input wide-character string

Then they shall attempt to convert the subject sequence to an unsigned integer, and return the result.

If *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a '+' or '−' sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7' only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a '+' or '−' sign, but not including an integer suffix. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* shall be permitted. If the value of *base* is 16, the wide-character codes 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first wide-character code that is not white space and is of the expected form. The subject sequence contains no wide-character codes if the input wide-character string is empty or consists entirely of white-space wide-character codes, or if the first wide-character code that is not white space is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and *base* is 0, the sequence of wide-character codes starting with the first digit shall be interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to the final wide-character string shall be stored in the object pointed to by *endptr*, provided that *endptr* is

not a null pointer.

CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX The *wcstoul*( ) function shall not change the setting of *errno* if successful.

Since 0, {ULONG_MAX}, and {ULLONG_MAX} are returned on error and 0 is also a valid return on success, an application wishing to check for error situations should set *errno* to 0, then call *wcstoul*( ) or *wcstoull*( ), then check *errno*.

RETURN VALUE

Upon successful completion, the *wcstoul*( ) and *wcstoull*( ) functions shall return the converted value, if any. If no conversion could be performed, 0 shall be returned and *errno* may be set to indicate the error. If the correct value is outside the range of representable values, {ULONG_MAX} or {ULLONG_MAX} respectively shall be returned and *errno* set to [ERANGE].

ERRORS

These functions shall fail if:

CX [EINVAL] The value of *base* is not supported.

[ERANGE] The value to be returned is not representable.

These functions may fail if:

CX [EINVAL] No conversion could be performed.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalpha( ), *scanf*( ), *wcstod*( ), *wcstol*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EINVAL] error condition is added for when the value of *base* is not supported.

In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The *wcstoul*() prototype is updated.
- The *wcstoull*() function is added.

50392 **NAME**

50393 wcstoull — convert a wide-character string to an unsigned long

50394 **SYNOPSIS**

50395 `#include <wchar.h>`

50396 `unsigned long long wcstoull(const wchar_t *restrict nptr,`
50397 `wchar_t **restrict endptr, int base);`

50398 **DESCRIPTION**

50399 Refer to *wcstoul*( ).

50400 **NAME**

50401 wcstoumax — convert wide-character string to integer type

50402 **SYNOPSIS**

50403 ```
#include <stddef.h>
```

50404 ```
#include <inttypes.h>
```

50405 ```
uintmax_t wcstoumax(const wchar_t *restrict nptr,
```

50406 ```
    wchar_t **restrict endptr, int base);
```

50407 **DESCRIPTION**

50408 Refer to *wcstoimax*( ).

50409 **NAME**

50410 wcswcs — find a wide substring (**LEGACY**)

50411 **SYNOPSIS**

50412 XSI
```
#include <wchar.h>
```

50413
```
wchar_t *wcswcs(const wchar_t *ws1, const wchar_t *ws2);
```

50414

50415 **DESCRIPTION**

50416 The *wcswcs*( ) function shall locate the first occurrence in the wide-character string pointed to by
50417 *ws1* of the sequence of wide-character codes (excluding the terminating null wide-character
50418 code) in the wide-character string pointed to by *ws2*.

50419 **RETURN VALUE**

50420 Upon successful completion, *wcswcs*( ) shall return a pointer to the located wide-character string
50421 or a null pointer if the wide-character string is not found.

50422 If *ws2* points to a wide-character string with zero length, the function shall return *ws1*.

50423 **ERRORS**

50424 No errors are defined.

50425 **EXAMPLES**

50426 None.

50427 **APPLICATION USAGE**

50428 This function was not included in the final ISO/IEC 9899:1990/Amendment 1:1995 (E).
50429 Application developers are strongly encouraged to use the *wcsstr*( ) function instead.

50430 **RATIONALE**

50431 None.

50432 **FUTURE DIRECTIONS**

50433 This function may be withdrawn in a future version.

50434 **SEE ALSO**

50435 *wcschr*( ), *wcsstr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

50436 **CHANGE HISTORY**

50437 First released in Issue 4. Derived from the MSE working draft.

50438 **Issue 5**

50439 Marked EX.

50440 **Issue 6**

50441 This function is marked LEGACY.

50442 **NAME**

50443 wcswidth — number of column positions of a wide-character string

50444 **SYNOPSIS**

50445 XSI
```
#include <wchar.h>

int wcswidth(const wchar_t *pwcs, size_t n);
```
50446

50447

50448 **DESCRIPTION**

50449 The *wcswidth*( ) function shall determine the number of column positions required for *n* wide-
50450 character codes (or fewer than *n* wide-character codes if a null wide-character code is
50451 encountered before *n* wide-character codes are exhausted) in the string pointed to by *pwcs*.

50452 **RETURN VALUE**

50453 The *wcswidth*( ) function either shall return 0 (if *pwcs* points to a null wide-character code), or
50454 return the number of column positions to be occupied by the wide-character string pointed to by
50455 *pwcs*, or return –1 (if any of the first *n* wide-character codes in the wide-character string pointed
50456 to by *pwcs* is not a printable wide-character code).

50457 **ERRORS**

50458 No errors are defined.

50459 **EXAMPLES**

50460 None.

50461 **APPLICATION USAGE**

50462 This function was removed from the final ISO / IEC 9899: 1990 / Amendment 1: 1995 (E), and the
50463 return value for a non-printable wide character is not specified.

50464 **RATIONALE**

50465 None.

50466 **FUTURE DIRECTIONS**

50467 None.

50468 **SEE ALSO**

50469 *wcwidth*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**, the Base Definitions
50470 volume of IEEE Std 1003.1-200x, Section 3.103, Column Position

50471 **CHANGE HISTORY**

50472 First released in Issue 4. Derived from the MSE working draft.

50473 **Issue 6**

50474 The Open Group Corrigendum U021 / 11 is applied. The function is marked as an extension.

50475 **NAME**

50476 wcsxfrm — wide-character string transformation

50477 **SYNOPSIS**

50478
```
#include <wchar.h>
```

50479
```
size_t wcsxfrm(wchar_t *restrict ws1, const wchar_t *restrict ws2,
50480     size_t n);
```

50481 **DESCRIPTION**

50482 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50483 conflict between the requirements described here and the ISO C standard is unintentional. This
50484 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50485 The *wcsxfrm*( ) function shall transform the wide-character string pointed to by *ws2* and place the
50486 resulting wide-character string into the array pointed to by *ws1*. The transformation shall be |
50487 such that if *wcscmp*( ) is applied to two transformed wide strings, it shall return a value greater |
50488 than, equal to, or less than 0, corresponding to the result of *wcscoll*( ) applied to the same two
50489 original wide-character strings. No more than *n* wide-character codes shall be placed into the |
50490 resulting array pointed to by *ws1*, including the terminating null wide-character code. If *n* is 0, |
50491 *ws1* is permitted to be a null pointer. If copying takes place between objects that overlap, the
50492 behavior is undefined.

50493 CX The *wcsxfrm*( ) function shall not change the setting of *errno* if successful.

50494 Since no return value is reserved to indicate an error, an application wishing to check for error |
50495 situations should set *errno* to 0, then call *wcsxfrm*( ), then check *errno*.

50496 **RETURN VALUE**

50497 The *wcsxfrm*( ) function shall return the length of the transformed wide-character string (not
50498 including the terminating null wide-character code). If the value returned is *n* or more, the
50499 contents of the array pointed to by *ws1* are unspecified. |

50500 CX On error, the *wcsxfrm*( ) function may set *errno*, but no return value is reserved to indicate an
50501 error.

50502 **ERRORS**

50503 The *wcsxfrm*( ) function may fail if:

50504 CX [EINVAL] The wide-character string pointed to by *ws2* contains wide-character codes
50505 outside the domain of the collating sequence.

50506 **EXAMPLES**

50507 None.

50508 **APPLICATION USAGE**

50509 The transformation function is such that two transformed wide-character strings can be ordered
50510 by *wcscmp*( ) as appropriate to collating sequence information in the program's locale (category
50511 *LC_COLLATE*).

50512 The fact that when *n* is 0 *ws1* is permitted to be a null pointer is useful to determine the size of
50513 the *ws1* array prior to making the transformation.

50514 **RATIONALE**

50515 None.

50516 **FUTURE DIRECTIONS**

50517 None.

50518 **SEE ALSO**

50519 *wcscmp*( ), *wcscoll*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

50520 **CHANGE HISTORY**

50521 First released in Issue 4. Derived from the MSE working draft.

50522 **Issue 5**

50523 Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

50524 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

50525 **Issue 6**

50526 In previous versions, this function was required to return –1 on error.

50527 Extensions beyond the ISO C standard are now marked.

50528 The following new requirements on POSIX implementations derive from alignment with the
50529 Single UNIX Specification:

50530 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
50531 added if no conversion could be performed.

50532 The *wcsxfrm*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

wctob — wide-character to single-byte conversion

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int wctob(wint_t c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wctob*( ) function shall determine whether *c* corresponds to a member of the extended character set whose character representation is a single byte when in the initial shift state.

The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

The *wctob*( ) function shall return EOF if *c* does not correspond to a character with length one in the initial shift state. Otherwise, it shall return the single-byte representation of that character as an **unsigned char** converted to **int**.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

btowc( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

NAME

wctomb — convert a wide-character code to a character

SYNOPSIS

```
#include <stdlib.h>

int wctomb(char *s, wchar_t wchar);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wctomb*( ) function shall determine the number of bytes needed to represent the character corresponding to the wide-character code whose value is *wchar* (including any change in the shift state). It shall store the character representation (possibly multiple bytes and any special bytes to change shift state) in the array object pointed to by *s* (if *s* is not a null pointer). At most {MB_CUR_MAX} bytes shall be stored. If *wchar* is 0, a null byte shall be stored, preceded by any shift sequence needed to restore the initial shift state, and *wctomb*( ) shall be left in the initial shift state.

CX The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a state-dependent encoding, this function shall be placed into its initial state by a call for which its character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a null pointer shall cause this function to return a non-zero value if encodings have state dependency, and 0 otherwise. Changing the *LC_CTYPE* category causes the shift state of this function to be unspecified.

The *wctomb*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *wctomb*( ).

RETURN VALUE

If *s* is a null pointer, *wctomb*( ) shall return a non-zero or 0 value, if character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *wctomb*( ) shall return –1 if the value of *wchar* does not correspond to a valid character, or return the number of bytes that constitute the character corresponding to the value of *wchar*.

In no case shall the value returned be greater than the value of the {MB_CUR_MAX} macro.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

50608 **SEE ALSO**

50609 *mblen*( ), *mbtowc*( ), *mbstowcs*( ), *wcstombs*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
50610 **<stdlib.h>**

50611 **CHANGE HISTORY**

50612 First released in Issue 4. Derived from the ANSI C standard.

50613 **Issue 6**

50614 Extensions beyond the ISO C standard are now marked.

50615 In the DESCRIPTION, a note about reentrancy and thread-safety is added.

50616 **NAME**

50617 wctrans — define character mapping

50618 **SYNOPSIS**

50619
```
#include <wctype.h>
```

50620
```
wctrans_t wctrans(const char *charclass);
```

50621 **DESCRIPTION**

50622 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50623 conflict between the requirements described here and the ISO C standard is unintentional. This
50624 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50625 The *wctrans*( ) function is defined for valid character mapping names identified in the current
50626 locale. The *charclass* is a string identifying a generic character mapping name for which codeset-
50627 specific information is required. The following character mapping names are defined in all
50628 locales: **tolower** and **toupper**.

50629 The function shall return a value of type **wctrans_t**, which can be used as the second argument
50630 to subsequent calls of *towctrans*( ). The *wctrans*( ) function shall determine values of **wctrans_t** |
50631 according to the rules of the coded character set defined by character mapping information in
50632 the program's locale (category *LC_CTYPE*). The values returned by *wctrans*( ) shall be valid until |
50633 a call to *setlocale*( ) that modifies the category *LC_CTYPE*. |

50634 **RETURN VALUE**

50635 CX The *wctrans*( ) function shall return 0 and may set *errno* to indicate the error if the given |
50636 character mapping name is not valid for the current locale (category *LC_CTYPE*); otherwise, it |
50637 shall return a non-zero object of type **wctrans_t** that can be used in calls to *towctrans*( ).

50638 **ERRORS**

50639 The *wctrans*( ) function may fail if:

50640 CX [EINVAL] The character mapping name pointed to by *charclass* is not valid in the current
50641 locale.

50642 **EXAMPLES**

50643 None.

50644 **APPLICATION USAGE**

50645 None.

50646 **RATIONALE**

50647 None.

50648 **FUTURE DIRECTIONS**

50649 None.

50650 **SEE ALSO**

50651 *towctrans*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**

50652 **CHANGE HISTORY**

50653 First released in Issue 5. Derived from ISO / IEC 9899: 1990 / Amendment 1: 1995 (E).

50654 **NAME**

50655 wctype — define character class

50656 **SYNOPSIS**

50657 `#include <wctype.h>`

50658 `wctype_t wctype(const char *property);`

50659 **DESCRIPTION**

50660 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50661 conflict between the requirements described here and the ISO C standard is unintentional. This
50662 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50663 The *wctype*( ) function is defined for valid character class names as defined in the current locale.
50664 The *property* argument is a string identifying a generic character class for which codeset-specific
50665 type information is required. The following character class names shall be defined in all locales:

50666 **alnum** **digit** **punct**
50667 **alpha** **graph** **space**
50668 **blank** **lower** **upper**
50669 **cntrl** **print** **xdigit**

50670 Additional character class names defined in the locale definition file (category *LC_CTYPE*) can
50671 also be specified.

50672 The function shall return a value of type **wctype_t**, which can be used as the second argument to
50673 subsequent calls of *iswctype*( ). The *wctype*( ) function shall determine values of **wctype_t**
50674 according to the rules of the coded character set defined by character type information in the
50675 program's locale (category *LC_CTYPE*). The values returned by *wctype*( ) shall be valid until a
50676 call to *setlocale*( ) that modifies the category *LC_CTYPE*.

50677 **RETURN VALUE**

50678 The *wctype*( ) function shall return 0 if the given character class name is not valid for the current
50679 locale (category *LC_CTYPE*); otherwise, it shall return an object of type **wctype_t** that can be
50680 used in calls to *iswctype*( ).

50681 **ERRORS**

50682 No errors are defined.

50683 **EXAMPLES**

50684 None.

50685 **APPLICATION USAGE**

50686 None.

50687 **RATIONALE**

50688 None.

50689 **FUTURE DIRECTIONS**

50690 None.

50691 **SEE ALSO**

50692 *iswctype*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**

50693 **CHANGE HISTORY**

50694 First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

50700 **NAME**

50701 wcwidth — number of column positions of a wide-character code

50702 **SYNOPSIS**

50703 XSI
```
#include <wchar.h>

int wcwidth(wchar_t wc);
```
50704

50705

50706 **DESCRIPTION**

50707 The *wcwidth*( ) function shall determine the number of column positions required for the wide
50708 character *wc*. The application shall ensure that the value of *wc* is a character representable as a
50709 **wchar_t**, and is a wide-character code corresponding to a valid character in the current locale. |

50710 **RETURN VALUE**

50711 The *wcwidth*( ) function shall either return 0 (if *wc* is a null wide-character code), or return the
50712 number of column positions to be occupied by the wide-character code *wc*, or return –1 (if *wc* |
50713 does not correspond to a printable wide-character code). |

50714 **ERRORS**

50715 No errors are defined.

50716 **EXAMPLES**

50717 None.

50718 **APPLICATION USAGE**

50719 This function was removed from the final ISO / IEC 9899: 1990 / Amendment 1: 1995 (E), and the
50720 return value for a non-printable wide character is not specified.

50721 **RATIONALE**

50722 None.

50723 **FUTURE DIRECTIONS**

50724 None.

50725 **SEE ALSO**

50726 *wcswidth*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

50727 **CHANGE HISTORY**

50728 First released as a World-wide Portability Interface in Issue 4. Derived from MSE working draft.

50729 **Issue 6**

50730 The Open Group Corrigendum U021 / 12 is applied. This function is marked as an extension.

50731 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

50732 **NAME**

50733 wmemchr — find a wide character in memory

50734 **SYNOPSIS**

50735 `#include <wchar.h>`

50736 `wchar_t *wmemchr(const wchar_t *ws, wchar_t wc, size_t n);`

50737 **DESCRIPTION**

50738 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50739 conflict between the requirements described here and the ISO C standard is unintentional. This
50740 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50741 The *wmemchr*( ) function shall locate the first occurrence of *wc* in the initial *n* wide characters of
50742 the object pointed to by *ws*. This function shall not be affected by locale and all **wchar_t** values |
50743 shall be treated identically. The null wide character and **wchar_t** values not corresponding to |
50744 valid characters shall not be treated specially. |

50745 If *n* is zero, the application shall ensure that *ws* is a valid pointer and the function behaves as if
50746 no valid occurrence of *wc* is found.

50747 **RETURN VALUE**

50748 The *wmemchr*( ) function shall return a pointer to the located wide character, or a null pointer if
50749 the wide character does not occur in the object.

50750 **ERRORS**

50751 No errors are defined.

50752 **EXAMPLES**

50753 None.

50754 **APPLICATION USAGE**

50755 None.

50756 **RATIONALE**

50757 None.

50758 **FUTURE DIRECTIONS**

50759 None.

50760 **SEE ALSO**

50761 *wmemcmp*( ), *wmemcpy*( ), *wmemmove*( ), *wmemset*( ), the Base Definitions volume of
50762 IEEE Std 1003.1-200x, **<wchar.h>**

50763 **CHANGE HISTORY**

50764 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50765 (E).

50766 **Issue 6**

50767 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

50768 **NAME**

50769 wmemcmp — compare wide characters in memory

50770 **SYNOPSIS**

50771 ```
#include <wchar.h>
```

50772 ```
int wmemcmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

50773 **DESCRIPTION**

50774 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50775 conflict between the requirements described here and the ISO C standard is unintentional. This
50776 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50777 The *wmemcmp*( ) function shall compare the first *n* wide characters of the object pointed to by
50778 *ws1* to the first *n* wide characters of the object pointed to by *ws2*. This function shall not be |
50779 affected by locale and all **wchar_t** values shall be treated identically. The null wide character and |
50780 **wchar_t** values not corresponding to valid characters shall not be treated specially. |

50781 If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function |
50782 shall behave as if the two objects compare equal. |

50783 **RETURN VALUE**

50784 The *wmemcmp*( ) function shall return an integer greater than, equal to, or less than zero,
50785 respectively, as the object pointed to by *ws1* is greater than, equal to, or less than the object
50786 pointed to by *ws2*.

50787 **ERRORS**

50788 No errors are defined.

50789 **EXAMPLES**

50790 None.

50791 **APPLICATION USAGE**

50792 None.

50793 **RATIONALE**

50794 None.

50795 **FUTURE DIRECTIONS**

50796 None.

50797 **SEE ALSO**

50798 *wmemchr*( ), *wmemcpy*( ), *wmemmove*( ), *wmemset*( ), the Base Definitions volume of
50799 IEEE Std 1003.1-200x, **<wchar.h>**

50800 **CHANGE HISTORY**

50801 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50802 (E).

50803 **Issue 6**

50804 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

50805 **NAME**

50806 wmemcpy — copy wide characters in memory

50807 **SYNOPSIS**

50808 ```
#include <wchar.h>
```

50809 ```
wchar_t *wmemcpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,
50810     size_t n);
```

50811 **DESCRIPTION**

50812 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50813 conflict between the requirements described here and the ISO C standard is unintentional. This
50814 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50815 The *wmemcpy*( ) function shall copy *n* wide characters from the object pointed to by *ws2* to the
50816 object pointed to by *ws1*. This function shall not be affected by locale and all **wchar_t** values |
50817 shall be treated identically. The null wide character and **wchar_t** values not corresponding to |
50818 valid characters shall not be treated specially. |

50819 If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function |
50820 shall copy zero wide characters. |

50821 **RETURN VALUE**

50822 The *wmemcpy*( ) function shall return the value of *ws1*.

50823 **ERRORS**

50824 No errors are defined.

50825 **EXAMPLES**

50826 None.

50827 **APPLICATION USAGE**

50828 None.

50829 **RATIONALE**

50830 None.

50831 **FUTURE DIRECTIONS**

50832 None.

50833 **SEE ALSO**

50834 *wmemchr*( ), *wmemcmp*( ), *wmemmove*( ), *wmemset*( ), the Base Definitions volume of
50835 IEEE Std 1003.1-200x, **<wchar.h>**

50836 **CHANGE HISTORY**

50837 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50838 (E).

50839 **Issue 6**

50840 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

50841 The *wmemcpy*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

50842 **NAME**

50843 wmemmove — copy wide characters in memory with overlapping areas

50844 **SYNOPSIS**

50845 ```
#include <wchar.h>
```

50846 ```
wchar_t *wmemmove(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

50847 **DESCRIPTION**

50848 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50849 conflict between the requirements described here and the ISO C standard is unintentional. This
50850 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50851 The *wmemmove*( ) function shall copy *n* wide characters from the object pointed to by *ws2* to the
50852 object pointed to by *ws1*. Copying shall take place as if the *n* wide characters from the object |
50853 pointed to by *ws2* are first copied into a temporary array of *n* wide characters that does not
50854 overlap the objects pointed to by *ws1* or *ws2*, and then the *n* wide characters from the temporary
50855 array are copied into the object pointed to by *ws1*.

50856 This function shall not be affected by locale and all **wchar_t** values shall be treated identically. |
50857 The null wide character and **wchar_t** values not corresponding to valid characters shall not be |
50858 treated specially. |

50859 If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function |
50860 shall copy zero wide characters. |

50861 **RETURN VALUE**

50862 The *wmemmove*( ) function shall return the value of *ws1*.

50863 **ERRORS**

50864 No errors are defined

50865 **EXAMPLES**

50866 None.

50867 **APPLICATION USAGE**

50868 None.

50869 **RATIONALE**

50870 None.

50871 **FUTURE DIRECTIONS**

50872 None.

50873 **SEE ALSO**

50874 *wmemchr*( ), *wmemcmp*( ), *wmemcpy*( ), *wmemset*( ), the Base Definitions volume of
50875 IEEE Std 1003.1-200x, **<wchar.h>**

50876 **CHANGE HISTORY**

50877 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50878 (E).

50879 **Issue 6**

50880 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

50881 **NAME**

50882 wmemset — set wide characters in memory

50883 **SYNOPSIS**

50884 `#include <wchar.h>`

50885 `wchar_t *wmemset(wchar_t *ws, wchar_t wc, size_t n);`

50886 **DESCRIPTION**

50887 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50888 conflict between the requirements described here and the ISO C standard is unintentional. This
50889 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50890 The *wmemset*( ) function shall copy the value of *wc* into each of the first *n* wide characters of the
50891 object pointed to by *ws*. This function shall not be affected by locale and all **wchar_t** values shall |
50892 be treated identically. The null wide character and **wchar_t** values not corresponding to valid |
50893 characters shall not be treated specially. |

50894 If *n* is zero, the application shall ensure that *ws* is a valid pointer, and the function shall copy |
50895 zero wide characters. |

50896 **RETURN VALUE**

50897 The *wmemset*( ) functions shall return the value of *ws*.

50898 **ERRORS**

50899 No errors are defined.

50900 **EXAMPLES**

50901 None.

50902 **APPLICATION USAGE**

50903 None.

50904 **RATIONALE**

50905 None.

50906 **FUTURE DIRECTIONS**

50907 None.

50908 **SEE ALSO**

50909 *wmemchr*( ), *wmemcmp*( ), *wmemcpy*( ), *wmemmove*( ), the Base Definitions volume of
50910 IEEE Std 1003.1-200x, **<wchar.h>**

50911 **CHANGE HISTORY**

50912 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50913 (E).

50914 **Issue 6**

50915 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

50916 **NAME**

50917 wordexp, wordfree — perform word expansions

50918 **SYNOPSIS**

50919 ```
#include <wordexp.h>
```

50920 ```
int wordexp(const char *restrict words, wordexp_t *restrict pwordexp,
```
50921 ```
    int flags);
```
50922 ```
void wordfree(wordexp_t *pwordexp);
```

50923 **DESCRIPTION**

50924 The *wordexp*( ) function shall perform word expansions as described in the Shell and Utilities |
50925 volume of IEEE Std 1003.1-200x, Section 2.6, Word Expansions, subject to quoting as in the Shell |
50926 and Utilities volume of IEEE Std 1003.1-200x, Section 2.2, Quoting, and place the list of expanded |
50927 words into the structure pointed to by *pwordexp*. |

50928 The *words* argument is a pointer to a string containing one or more words to be expanded. The
50929 expansions shall be the same as would be performed by the command line interpreter if *words*
50930 were the part of a command line representing the arguments to a utility. Therefore, the
50931 application shall ensure that *words* does not contain an unquoted <newline> or any of the
50932 unquoted shell special characters `'|'`, `'&'`, `';'`, `'<'`, `'>'` except in the context of command
50933 substitution as specified in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.3,
50934 Command Substitution. It also shall not contain unquoted parentheses or braces, except in the
50935 context of command or variable substitution. The application shall ensure that every member of
50936 *words* which it expects to have expanded by *wordexp*( ) does not contain an unquoted initial
50937 comment character. The application shall also ensure that any words which it intends to be
50938 ignored (because they begin or continue a comment) are deleted from *words*. If the argument
50939 *words* contains an unquoted comment character (number sign) that is the beginning of a token,
50940 *wordexp*( ) shall either treat the comment character as a regular character, or interpret it as a
50941 comment indicator and ignore the remainder of *words*.

50942 The structure type **wordexp_t** is defined in the **<wordexp.h>** header and includes at least the |
50943 following members: |

50944

50945

Member Type	Member Name	Description
size_t	*we_wordc*	Count of words matched by *words*.
char **	*we_wordv*	Pointer to list of expanded words.
size_t	*we_offs*	Slots to reserve at the beginning of *pwordexp->we_wordv*.

50949 The *wordexp*( ) function shall store the number of generated words into *pwordexp->we_wordc* and |
50950 a pointer to a list of pointers to words in *pwordexp->we_wordv*. Each individual field created |
50951 during field splitting (see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.5, |
50952 Field Splitting) or pathname expansion (see the Shell and Utilities volume of |
50953 IEEE Std 1003.1-200x, Section 2.6.6, Pathname Expansion) shall be a separate word in the |
50954 *pwordexp->we_wordv* list. The words shall be in order as described in the Shell and Utilities |
50955 volume of IEEE Std 1003.1-200x, Section 2.6, Word Expansions. The first pointer after the last |
50956 word pointer shall be a null pointer. The expansion of special parameters described in the Shell |
50957 and Utilities volume of IEEE Std 1003.1-200x, Section 2.5.2, Special Parameters is unspecified. |

50958 It is the caller's responsibility to allocate the storage pointed to by *pwordexp*. The *wordexp*( ) |
50959 function shall allocate other space as needed, including memory pointed to by *pwordexp-* |
50960 *>we_wordv*. The *wordfree*( ) function frees any memory associated with *pwordexp* from a previous |
50961 call to *wordexp*( ).

The *flags* argument is used to control the behavior of *wordexp*(). The value of *flags* is the bitwise-inclusive OR of zero or more of the following constants, which are defined in **<wordexp.h>**:

WRDE_APPEND — Append words generated to the ones from a previous call to *wordexp*().

WRDE_DOOFFS — Make use of *pwordexp->we_offs*. If this flag is set, *pwordexp->we_offs* is used to specify how many null pointers to add to the beginning of *pwordexp->we_wordv*. In other words, *pwordexp->we_wordv* shall point to *pwordexp->we_offs* null pointers, followed by *pwordexp->we_wordc* word pointers, followed by a null pointer.

WRDE_NOCMD — If the implementation supports the utilities defined in the Shell and Utilities volume of IEEE Std 1003.1-200x, fail if command substitution, as specified in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.3, Command Substitution, is requested.

WRDE_REUSE — The *pwordexp* argument was passed to a previous successful call to *wordexp*(), and has not been passed to *wordfree*(). The result shall be the same as if the application had called *wordfree*() and then called *wordexp*() without WRDE_REUSE.

WRDE_SHOWERR — Do not redirect *stderr* to **/dev/null**.

WRDE_UNDEF — Report error on an attempt to expand an undefined shell variable.

The WRDE_APPEND flag can be used to append a new set of words to those generated by a previous call to *wordexp*(). The following rules apply to applications when two or more calls to *wordexp*() are made with the same value of *pwordexp* and without intervening calls to *wordfree*():

1. The first such call shall not set WRDE_APPEND. All subsequent calls shall set it.
2. All of the calls shall set WRDE_DOOFFS, or all shall not set it.
3. After the second and each subsequent call, *pwordexp->we_wordv* shall point to a list containing the following:
   a. Zero or more null pointers, as specified by WRDE_DOOFFS and *pwordexp->we_offs*
   b. Pointers to the words that were in the *pwordexp->we_wordv* list before the call, in the same order as before
   c. Pointers to the new words generated by the latest call, in the specified order
4. The count returned in *pwordexp->we_wordc* shall be the total number of words from all of the calls.
5. The application can change any of the fields after a call to *wordexp*(), but if it does it shall reset them to the original value before a subsequent call, using the same *pwordexp* value, to *wordfree*() or *wordexp*() with the WRDE_APPEND or WRDE_REUSE flag.

If the implementation supports the utilities defined in the Shell and Utilities volume of IEEE Std 1003.1-200x, and *words* contains an unquoted character—<newline>, `'|'`, `'&'`, `';'`, `'<'`, `'>'`, `'('`, `')'`, `'{'`, `'}'`—in an inappropriate context, *wordexp*() shall fail, and the number of expanded words shall be 0.

Unless WRDE_SHOWERR is set in *flags*, *wordexp*() shall redirect *stderr* to **/dev/null** for any utilities executed as a result of command substitution while expanding *words*. If WRDE_SHOWERR is set, *wordexp*() may write messages to *stderr* if syntax errors are detected while expanding *words*.

The application shall ensure that if WRDE_DOOFFS is set, then *pwordexp->we_offs* has the same value for each *wordexp*( ) call and *wordfree*( ) call using a given *pwordexp*.

The following constants are defined as error return values:

WRDE_BADCHAR	One of the unquoted characters—<newline>, `'|'`, `'&'`, `';'`, `'<'`, `'>'`, `'('`, `')'`, `'{'`, `'}'`—appears in *words* in an inappropriate context.
WRDE_BADVAL	Reference to undefined shell variable when WRDE_UNDEF is set in *flags*.
WRDE_CMDSUB	Command substitution requested when WRDE_NOCMD was set in flags.
WRDE_NOSPACE	Attempt to allocate memory failed.
WRDE_SYNTAX	Shell syntax error, such as unbalanced parentheses or unterminated string.

RETURN VALUE

Upon successful completion, *wordexp*( ) shall return 0. Otherwise, a non-zero value, as described in **<wordexp.h>**, shall be returned to indicate an error. If *wordexp*( ) returns the value WRDE_NOSPACE, then *pwordexp->we_wordc* and *pwordexp->we_wordv* shall be updated to reflect any words that were successfully expanded. In other cases, they shall not be modified.

The *wordfree*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *wordexp*( ) function is intended to be used by an application that wants to do all of the shell's expansions on a word or words obtained from a user. For example, if the application prompts for a filename (or list of filenames) and then uses *wordexp*( ) to process the input, the user could respond with anything that would be valid as input to the shell.

The WRDE_NOCMD flag is provided for applications that, for security or other reasons, want to prevent a user from executing shell commands. Disallowing unquoted shell special characters also prevents unwanted side effects, such as executing a command or writing a file.

RATIONALE

This function was included as an alternative to *glob*( ). There had been continuing controversy over exactly what features should be included in *glob*( ). It is hoped that by providing *wordexp*( ) (which provides all of the shell word expansions, but which may be slow to execute) and *glob*( ) (which is faster, but which only performs pathname expansion, without tilde or parameter expansion) this will satisfy the majority of applications.

While *wordexp*( ) could be implemented entirely as a library routine, it is expected that most implementations run a shell in a subprocess to do the expansion.

Two different approaches have been proposed for how the required information might be presented to the shell and the results returned. They are presented here as examples.

One proposal is to extend the *echo* utility by adding a **−q** option. This option would cause *echo* to add a backslash before each backslash and <blank> that occurs within an argument. The *wordexp*( ) function could then invoke the shell as follows:

```
(void) strcpy(buffer, "echo -q");
(void) strcat(buffer, words);
if ((flags & WRDE_SHOWERR) == 0)
```

```
    (void) strcat(buffer, "2>/dev/null");
f = popen(buffer, "r");
```

The *wordexp*() function would read the resulting output, remove unquoted backslashes, and break into words at unquoted <blank>s. If the WRDE_NOCMD flag was set, *wordexp*() would have to scan *words* before starting the subshell to make sure that there would be no command substitution. In any case, it would have to scan *words* for unquoted special characters.

Another proposal is to add the following options to *sh*:

–w *wordlist*
This option provides a wordlist expansion service to applications. The words in *wordlist* shall be expanded and the following written to standard output:

1. The count of the number of words after expansion, in decimal, followed by a null byte
2. The number of bytes needed to represent the expanded words (not including null separators), in decimal, followed by a null byte
3. The expanded words, each terminated by a null byte

If an error is encountered during word expansion, *sh* exits with a non-zero status after writing the former to report any words successfully expanded

–P Run in ''protected'' mode. If specified with the **–w** option, no command substitution shall be performed.

With these options, *wordexp*() could be implemented fairly simply by creating a subprocess using *fork*() and executing *sh* using the line:

```
execl(<shell path>, "sh", "-P", "-w", words, (char *)0);
```

after directing standard error to **/dev/null**.

It seemed objectionable for a library routine to write messages to standard error, unless explicitly requested, so *wordexp*() is required to redirect standard error to **/dev/null** to ensure that no messages are generated, even for commands executed for command substitution. The WRDE_SHOWERR flag can be specified to request that error messages be written.

The WRDE_REUSE flag allows the implementation to avoid the expense of freeing and reallocating memory, if that is possible. A minimal implementation can call *wordfree*() when WRDE_REUSE is set.

FUTURE DIRECTIONS

None.

SEE ALSO

fnmatch(), *glob*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<wordexp.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

Moved from POSIX2 C-language Binding to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

The **restrict** keyword is added to the *wordexp*() prototype for alignment with the ISO / IEC 9899: 1999 standard.

51091 **NAME**

51092 wprintf — print formatted wide-character output

51093 **SYNOPSIS**

51094 `#include <stdio.h>`

51095 `#include <wchar.h>`

51096 `int wprintf(const wchar_t *restrict format, ...);` |

51097 **DESCRIPTION** |

51098 Refer to *fwprintf*( ).

NAME

pwrite, write — write on a file

SYNOPSIS

```
#include <unistd.h>

ssize_t pwrite(int fildes, const void *buf, size_t nbyte,
    off_t offset);
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

DESCRIPTION

The *write*( ) function shall attempt to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes*.

Before any action described below is taken, and if *nbyte* is zero and the file is a regular file, the *write*( ) function may detect and return errors as described below. In the absence of errors, or if error detection is not performed, the *write*( ) function shall return zero and have no other results. If *nbyte* is zero and the file is not a regular file, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data shall proceed from the position in the file indicated by the file offset associated with *fildes*. Before successful return from *write*( ), the file offset shall be incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file shall be set to this file offset.

On a file not capable of seeking, writing shall always take place starting at the current position. The value of a file offset associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file offset shall be set to the end of the file prior to each write and no intervening file modification operation shall occur between changing the file offset and the write operation.

XSI If a *write*( ) requests that more bytes be written than there is room for (for example, the process' file size limit or the physical end of a medium), only as many bytes as there is room for shall be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes would give a failure return (except as noted below).

XSI If the request would cause the file size to exceed the soft file size limit for the process and there is no room for any bytes to be written, the request shall fail and the implementation shall generate the SIGXFSZ signal for the thread.

If *write*( ) is interrupted by a signal before it writes any data, it shall return –1 with *errno* set to [EINTR].

If *write*( ) is interrupted by a signal after it successfully writes some data, it shall return the number of bytes written.

If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation-defined.

After a *write*( ) to a regular file has successfully returned:

- Any successful *read*( ) from each byte position in the file that was modified by that write shall return the data specified by the *write*( ) for that position until such byte positions are again modified.
- Any subsequent successful *write*( ) to the same byte position in the file shall overwrite that file data.

Write requests to a pipe or FIFO shall be handled in the same way as a regular file with the following exceptions:

- There is no file offset associated with a pipe, hence each write request shall append to the end of the pipe.
- Write requests of {PIPE_BUF} bytes or less shall not be interleaved with data from other processes doing writes on the same pipe. Writes of greater than {PIPE_BUF} bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the O_NONBLOCK flag of the file status flags is set.
- If the O_NONBLOCK flag is clear, a write request may cause the thread to block, but on normal completion it shall return *nbyte*.
- If the O_NONBLOCK flag is set, *write*( ) requests shall be handled differently, in the following ways:
  - The *write*( ) function shall not block the thread.
  - A write request for {PIPE_BUF} or fewer bytes shall have the following effect: if there is sufficient space available in the pipe, *write*( ) shall transfer all the data and return the number of bytes requested. Otherwise, *write*( ) shall transfer no data and return –1 with *errno* set to [EAGAIN].
  - A write request for more than {PIPE_BUF} bytes shall cause one of the following:
    - When at least one byte can be written, transfer what it can and return the number of bytes written. When all data previously written to the pipe is read, it shall transfer at least {PIPE_BUF} bytes.
    - When no data can be written, transfer no data, and return –1 with *errno* set to [EAGAIN].

When attempting to write to a file descriptor (other than a pipe or FIFO) that supports non-blocking writes and cannot accept the data immediately:

- If the O_NONBLOCK flag is clear, *write*( ) shall block the calling thread until the data can be accepted.
- If the O_NONBLOCK flag is set, *write*( ) shall not block the thread. If some data can be written without blocking the thread, *write*( ) shall write what it can and return the number of bytes written. Otherwise, it shall return –1 and set *errno* to [EAGAIN].

Upon successful completion, where *nbyte* is greater than 0, *write*( ) shall mark for update the *st_ctime* and *st_mtime* fields of the file, and if the file is a regular file, the S_ISUID and S_ISGID bits of the file mode may be cleared.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with *fildes*.

If *fildes* refers to a socket, *write*( ) shall be equivalent to *send*( ) with no flags set.

SIO If the O_DSYNC bit has been set, write I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion.

If the O_SYNC bit has been set, write I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

SHM If *fildes* refers to a shared memory object, the result of the *write*( ) function is unspecified.

TYM If *fildes* refers to a typed memory object, the result of the *write*( ) function is unspecified.

XSR If *fildes* refers to a STREAM, the operation of *write*() shall be determined by the values of the minimum and maximum *nbyte* range (packet size) accepted by the STREAM. These values are determined by the topmost STREAM module. If *nbyte* falls within the packet size range, *nbyte* bytes shall be written. If *nbyte* does not fall within the range and the minimum packet size value is 0, *write*() shall break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, *write*() shall fail with *errno* set to [ERANGE]. Writing a zero-length buffer (*nbyte* is 0) to a STREAMS device sends 0 bytes with 0 returned. However, writing a zero-length buffer to a STREAMS-based pipe or FIFO sends no message and 0 is returned. The process may issue I_SWROPT *ioctl*() to enable zero-length messages to be sent across the pipe or FIFO.

When writing to a STREAM, data messages are created with a priority band of 0. When writing to a STREAM that is not a pipe or FIFO:

- If O_NONBLOCK is clear, and the STREAM cannot accept data (the STREAM write queue is full due to internal flow control conditions), *write*() shall block until data can be accepted.
- If O_NONBLOCK is set and the STREAM cannot accept data, *write*() shall return –1 and set *errno* to [EAGAIN].
- If O_NONBLOCK is set and part of the buffer has been written while a condition in which the STREAM cannot accept additional data occurs, *write*() shall terminate and return the number of bytes written.

In addition, *write*() shall fail if the STREAM head has processed an asynchronous error before the call. In this case, the value of *errno* does not reflect the result of *write*(), but reflects the prior error.

XSI The *pwrite*() function shall be equivalent to *write*(), except that it writes into a given position without changing the file pointer. The first three arguments to *pwrite*() are the same as *write*() with the addition of a fourth argument offset for the desired position inside the file.

RETURN VALUE

XSI Upon successful completion, *write*() and *pwrite*() shall return the number of bytes actually written to the file associated with *fildes*. This number shall never be greater than *nbyte*. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

XSI The *write*() and *pwrite*() functions shall fail if:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the thread would be delayed in the *write*() operation.

[EBADF] The *fildes* argument is not a valid file descriptor open for writing.

[EFBIG] XSI An attempt was made to write a file that exceeds the implementation-defined maximum file size or the process' file size limit, and there was no room for any bytes to be written.

[EFBIG] The file is a regular file, *nbyte* is greater than 0, and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildes*.

[EINTR] The write operation was terminated due to the receipt of a signal, and no data was transferred.

[EIO] The process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor

blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.

[ENOSPC] There was no free space remaining on the device containing the file.

[EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by any process, or that only has one end open. A SIGPIPE signal shall also be sent to the thread.

XSR [ERANGE] The transfer request size was outside the range supported by the STREAMS file associated with *fildes*.

The *write*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]
The file descriptor is for a socket, is marked O_NONBLOCK, and write would block.

[ECONNRESET] A write was attempted on a socket that is not connected.

[EPIPE] A write was attempted on a socket that is shut down for writing, or is no longer connected. In the latter case, if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.

XSI The *write*( ) and *pwrite*( ) functions may fail if: |

XSR [EINVAL] The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

[EIO] A physical I/O error has occurred.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

XSR [ENXIO] A hangup occurred on the STREAM being written to.

XSR A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, *errno* is set to the value included in the error message.

The *write*( ) function may fail if:

[EACCES] A write was attempted on a socket and the calling process does not have appropriate privileges.

[ENETDOWN] A write was attempted on a socket and the local network interface used to reach the destination is down.

[ENETUNREACH] A write was attempted on a socket and no route to the network is present.

XSI The *pwrite*( ) function shall fail and the file pointer remain unchanged if: |

XSI [EINVAL] The *offset* argument is invalid. The value is negative.

XSI [ESPIPE] *fildes* is associated with a pipe or FIFO.

EXAMPLES

Writing from a Buffer

The following example writes data from the buffer pointed to by *buf* to the file associated with the file descriptor *fd*.

```
#include <sys/types.h>
#include <string.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_written;
int fd;
...
strcpy(buf, "This is a test\n");
nbytes = strlen(buf);

bytes_written = write(fd, buf, nbytes);
...
```

APPLICATION USAGE

None.

RATIONALE

See also the RATIONALE section in *read*( ).

An attempt to write to a pipe or FIFO has several major characteristics:

- *Atomic/non-atomic*: A write is atomic if the whole amount written in one operation is not interleaved with data from any other process. This is useful when there are multiple writers sending data to a single reader. Applications need to know how large a write request can be expected to be performed atomically. This maximum is called {PIPE_BUF}. This volume of IEEE Std 1003.1-200x does not say whether write requests for more than {PIPE_BUF} bytes are atomic, but requires that writes of {PIPE_BUF} or fewer bytes shall be atomic.

- *Blocking/immediate*: Blocking is only possible with O_NONBLOCK clear. If there is enough space for all the data requested to be written immediately, the implementation should do so. Otherwise, the process may block; that is, pause until enough space is available for writing. The effective size of a pipe or FIFO (the maximum amount that can be written in one operation without blocking) may vary dynamically, depending on the implementation, so it is not possible to specify a fixed value for it.

- *Complete/partial/deferred*: A write request:

```
int fildes;
size_t nbyte;
ssize_t ret;
char *buf;

ret = write(fildes, buf, nbyte);
```

  may return:

  complete *ret=nbyte*

  partial *ret<nbyte*

  This shall never happen if *nbyte*≤{PIPE_BUF}. If it does happen (with *nbyte*>{PIPE_BUF}), this volume of IEEE Std 1003.1-200x does not guarantee

atomicity, even if *ret*≤{PIPE_BUF}, because atomicity is guaranteed according to the amount *requested*, not the amount *written*.

deferred: *ret*=−1, *errno*=[EAGAIN]

This error indicates that a later request may succeed. It does not indicate that it *shall* succeed, even if *nbyte*≤{PIPE_BUF}, because if no process reads from the pipe or FIFO, the write never succeeds. An application could usefully count the number of times [EAGAIN] is caused by a particular value of *nbyte*>{PIPE_BUF} and perhaps do later writes with a smaller value, on the assumption that the effective size of the pipe may have decreased.

Partial and deferred writes are only possible with O_NONBLOCK set.

The relations of these properties are shown in the following tables:

Write to a Pipe or FIFO with O_NONBLOCK *clear*			
Immediately Writable:	**None**	**Some**	*nbyte*
nbyte≤{PIPE_BUF}	Atomic blocking *nbyte*	Atomic blocking *nbyte*	Atomic immediate *nbyte*
nbyte>{PIPE_BUF}	Blocking *nbyte*	Blocking *nbyte*	Blocking *nbyte*

If the O_NONBLOCK flag is clear, a write request shall block if the amount writable immediately is less than that requested. If the flag is set (by *fcntl*( )), a write request shall never block.

Write to a Pipe or FIFO with O_NONBLOCK *set*			
Immediately Writable:	**None**	**Some**	*nbyte*
nbyte≤{PIPE_BUF}	−1, [EAGAIN]	−1, [EAGAIN]	Atomic *nbyte*
nbyte>{PIPE_BUF}	−1, [EAGAIN]	<*nbyte* or −1, [EAGAIN]	≤*nbyte* or −1, [EAGAIN]

There is no exception regarding partial writes when O_NONBLOCK is set. With the exception of writing to an empty pipe, this volume of IEEE Std 1003.1-200x does not specify exactly when a partial write is performed since that would require specifying internal details of the implementation. Every application should be prepared to handle partial writes when O_NONBLOCK is set and the requested amount is greater than {PIPE_BUF}, just as every application should be prepared to handle partial writes on other kinds of file descriptors.

The intent of forcing writing at least one byte if any can be written is to assure that each write makes progress if there is any room in the pipe. If the pipe is empty, {PIPE_BUF} bytes must be written; if not, at least some progress must have been made.

Where this volume of IEEE Std 1003.1-200x requires −1 to be returned and *errno* set to [EAGAIN], most historical implementations return zero (with the O_NDELAY flag set, which is the historical predecessor of O_NONBLOCK, but is not itself in this volume of IEEE Std 1003.1-200x). The error indications in this volume of IEEE Std 1003.1-200x were chosen so that an application can distinguish these cases from end-of-file. While *write*( ) cannot receive an indication of end-of-file, *read*( ) can, and the two functions have similar return values. Also, some existing systems (for example, Eighth Edition) permit a write of zero bytes to mean that the reader should get an end-of-file indication; for those systems, a return value of zero from *write*( ) indicates a successful write of an end-of-file indication.

Implementations are allowed, but not required, to perform error checking for *write*( ) requests of zero bytes.

The concept of a {PIPE_MAX} limit (indicating the maximum number of bytes that can be written to a pipe in a single operation) was considered, but rejected, because this concept would unnecessarily limit application writing.

See also the discussion of O_NONBLOCK in *read*( ).

Writes can be serialized with respect to other reads and writes. If a *read*( ) of file data can be proven (by any means) to occur after a *write*( ) of the data, it must reflect that *write*( ), even if the calls are made by different processes. A similar requirement applies to multiple write operations to the same file position. This is needed to guarantee the propagation of data from *write*( ) calls to subsequent *read*( ) calls. This requirement is particularly significant for networked file systems, where some caching schemes violate these semantics.

Note that this is specified in terms of *read*( ) and *write*( ). The XSI extensions *readv*( ) and *writev*( ) also obey these semantics. A new ''high-performance'' write analog that did not follow these serialization requirements would also be permitted by this wording. This volume of IEEE Std 1003.1-200x is also silent about any effects of application-level caching (such as that done by *stdio*).

This volume of IEEE Std 1003.1-200x does not specify the value of the file offset after an error is returned; there are too many cases. For programming errors, such as [EBADF], the concept is meaningless since no file is involved. For errors that are detected immediately, such as [EAGAIN], clearly the pointer should not change. After an interrupt or hardware error, however, an updated value would be very useful and is the behavior of many implementations.

This volume of IEEE Std 1003.1-200x does not specify behavior of concurrent writes to a file from multiple processes. Applications should use some form of concurrency control.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *creat*( ), *dup*( ), *fcntl*( ), *getrlimit*( ), *lseek*( ), *open*( ), *pipe*( ), *ulimit*( ), *writev*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<stropts.h>**, **<sys/uio.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

The *pwrite*( ) function is added.

Issue 6

The DESCRIPTION states that the *write*( ) function does not block the thread. Previously this said ''process'' rather than ''thread''.

The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are marked as part of the XSI STREAMS Option Group.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION now states that if *write*( ) is interrupted by a signal after it has successfully written some data, it returns the number of bytes written. In earlier versions of this volume of IEEE Std 1003.1-200x, it was optional whether *write*( ) returned the number of bytes written, or whether it returned –1 with *errno* set to [EINTR]. This is a FIPS requirement.
- The following changes are made to support large files:
  - For regular files, no data transfer occurs past the offset maximum established in the open file description associated with the *fildes*.
  - A second [EFBIG] error condition is added.
- The [EIO] error condition is added.
- The [EPIPE] error condition is added for when a pipe has only one end open.
- The [ENXIO] optional error condition is added.

Text referring to sockets is added to the DESCRIPTION.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effect of reading zero bytes is clarified.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that *write*( ) results are unspecified for typed memory objects.

The following error conditions are added for operations on sockets: [EAGAIN], [EWOULDBLOCK], [ECONNRESET], [ENOTCONN], and [EPIPE].

The [EIO] error is changed to ‘‘may fail’’.

The [ENOBUFS] error is added for sockets.

The following error conditions are added for operations on sockets: [EACCES], [ENETDOWN], and [ENETUNREACH].

The *writev*( ) function is split out into a separate reference page.

NAME

writev — write a vector

SYNOPSIS

XSI

```
#include <sys/uio.h>

ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);
```

DESCRIPTION

The *writev*( ) function shall be equivalent to *write*( ), except as described below. The *writev*( ) function shall gather output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*–1]. The *iovcnt* argument is valid if greater than 0 and less than or equal to {IOV_MAX}, defined in **<limits.h>**.

Each *iovec* entry specifies the base address and length of an area in memory from which data should be written. The *writev*( ) function shall always write a complete area before proceeding to the next.

If *fildes* refers to a regular file and all of the *iov_len* members in the array pointed to by *iov* are 0, *writev*( ) shall return 0 and have no other effect. For other file types, the behavior is unspecified.

If the sum of the *iov_len* values is greater than {SSIZE_MAX}, the operation shall fail and no data shall be transferred.

RETURN VALUE

Upon successful completion, *writev*( ) shall return the number of bytes actually written. Otherwise, it shall return a value of –1, the file-pointer shall remain unchanged, and *errno* shall be set to indicate an error.

ERRORS

Refer to *write*( ).

In addition, the *writev*( ) function shall fail if:

[EINVAL] The sum of the *iov_len* values in the *iov* array would overflow an **ssize_t**.

The *writev*( ) function may fail and set *errno* to:

[EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than {IOV_MAX}.

EXAMPLES

Writing Data from an Array

The following example writes data from the buffers specified by members of the *iov* array to the file associated with the file descriptor *fd*.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
...
ssize_t bytes_written;
int fd;
char *buf0 = "short string\n";
char *buf1 = "This is a longer string\n";
char *buf2 = "This is the longest string in this example\n";
int iovcnt;
struct iovec iov[3];
```

```
51461       iov[0].iov_base = buf0;
51462       iov[0].iov_len = strlen(buf0);
51463       iov[1].iov_base = buf1;
51464       iov[1].iov_len = strlen(buf1);
51465       iov[2].iov_base = buf2;
51466       iov[2].iov_len = strlen(buf2);
51467       ...
51468       iovcnt = sizeof(iov) / sizeof(struct iovec);

51469       bytes_written = writev(fd, iov, iovcnt);
51470       ...
```

51471 **APPLICATION USAGE**

51472 None.

51473 **RATIONALE**

51474 Refer to *write*( ).

51475 **FUTURE DIRECTIONS**

51476 None.

51477 **SEE ALSO**

51478 *readv*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<sys/uio.h>**

51479 **CHANGE HISTORY**

51480 First released in Issue 4, Version 2.

51481 **Issue 6**

51482 Split out from the *write*( ) reference page.

51483 **NAME**

51484 wscanf — convert formatted wide-character input

51485 **SYNOPSIS**

51486 `#include <stdio.h>`

51487 `#include <wchar.h>`

51488 `int wscanf(const wchar_t *restrict format, ... );` |

51489 **DESCRIPTION** |

51490 Refer to *fwscanf*( ).

NAME

y0, y1, yn — Bessel functions of the second kind

SYNOPSIS

XSI

```
#include <math.h>

double y0(double x);
double y1(double x);
double yn(int n, double x);
```

DESCRIPTION

The *y0*(), *y1*(), and *yn*() functions shall compute Bessel functions of *x* of the second kind of orders 0, 1, and *n*, respectively.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the relevant Bessel value of *x* of the second kind.

If *x* is NaN, NaN shall be returned.

If the *x* argument to these functions is negative, −HUGE_VAL or NaN shall be returned, and a domain error may occur.

If *x* is 0.0, −HUGE_VAL shall be returned and a range error may occur.

If the correct result would cause underflow, 0.0 shall be returned and a range error may occur.

If the correct result would cause overflow, −HUGE_VAL or 0.0 shall be returned and a range error may occur.

ERRORS

These functions may fail if:

Domain Error	The value of *x* is negative.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
Range Error	The value of *x* is 0.0, or the correct result would cause overflow.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.
Range Error	The value of *x* is too large in magnitude, or the correct result would cause underflow.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *isnan*( ), *j0*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The RETURN VALUE and ERRORS sections are reworked for alignment of the error handling with the ISO/IEC 9899:1999 standard.