Rationale (Informative)

Part A:
Base Definitions

The Open Group

Appendix A

Rationale for Base Definitions

A.1 Introduction

A.1.1 Scope

IEEE Std. 1003.1-200x is one of a family of standards known as POSIX. The family of standards extends to many topics; IEEE Std. 1003.1-200x is known as POSIX.1 and consists of both operating system interfaces and shell and utilities.

Scope of IEEE Std. 1003.1-200x

The (paraphrased) goals of this development were to produce a single common revision to the overlapping POSIX.1 and POSIX.2 standards, and the Single UNIX Specification, Version 2. As such, the scope of the revision includes the scopes of the original documents merged.

Since the revision includes merging the Base volumes of the Single UNIX Specification, many features that were previously not *adopted* into earlier revisions of POSIX.1 and POSIX.2 are now included in IEEE Std. 1003.1-200x. In most cases, these additions are part of the XSI extension; in other cases the standard developers decided that now was the time to migrate these to the base standard.

The Single UNIX Specification programming environment provides a broad-based functional set of interfaces to support the porting of existing UNIX applications and the development of new applications. The environment also supports a rich set of tools for application development.

The majority of the obsolescent material from the existing POSIX.1 and POSIX.2 standards, and material marked LEGACY from The Open Group's Base specifications, has been removed in this revision.

The following IEEE Standards have been added to the base documents in this revision:

- IEEE Std. 1003.1d-1999
- IEEE Std. 1003.1j-2000
- IEEE Std. 1003.1q-2000
- IEEE P1003.1a draft standard
- IEEE Std. 1003.2d-1994
- IEEE P1003.2b draft standard
- Selected parts of IEEE Std. 1003.1g-2000

Only selected parts of IEEE Std. 1003.1g-2000 were included. This was because there is much duplication between the XNS, Issue 5.2 specification (another base document) and the material from IEEE Std. 1003.1g-2000, the former document being aligned with the latest networking specifications for IPv6. Only the following sections of IEEE Std. 1003.1g-2000 were considered for inclusion:

- General terms related to sockets (clause 2.2.2)

- Socket concepts (clauses 5.1 through 5.3 inclusive)
- The *pselect*( ) function (clauses 6.2.2.1 and 6.2.3)
- The *isfdtype*( ) function (clause 5.4.8)
- The **<sys/select.h>** header (clause 6.2)

The following were requirements on IEEE Std. 1003.1-200x:

- Backward-compatibility

  It was agreed that there should be no breakage of functionality in the existing base documents. This requirement was tempered by changes introduced due to interpretations and corrigenda on the base documents, and any changes introduced in the ISO / IEC 9899: 1999 standard (C Language).

- Architecture and n-bit neutral

  The common standard should not make any implicit assumptions about the system architecture or size of data types; for example, previously some 32-bit implicit assumptions had crept into the standards.

- Extensibility

  It should be possible to extend the common standard without breaking backward-compatibility. For example, the name space should be reserved and structured to avoid duplication of names between the standard and extensions to it.

POSIX.1 and the ISO C standard

Previous revisions of POSIX.1 built upon the ISO C standard by reference only. This revision takes a different approach.

The standard developers believed it essential for a programmer to have a single complete reference place, but recognized that deference to the formal standard had to be addressed for the the duplicate interface definitions between the ISO C standard and the Single UNIX Specification.

It was agreed that where an interface has a version in the ISO C standard, the DESCRIPTION section should describe the relationship to the ISO C standard and markings should be added as appropriate to show where the ISO C standard has been extended in the text.

The following block of text was added to each reference page affected:

> *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.*

and each page was parsed for additions beyond the ISO C standard (that is, including both POSIX and UNIX extensions), and these extensions are marked as CX extensions (for C Extensions).

A.1.2 Conformance

See Section A.2 (on page 3317).

A.1.3 Normative References

There is no additional rationale for this section.

A.1.4 Terminology

The meanings specified in IEEE Std. 1003.1-200x for the words *shall*, *should*, and *may* are mandated by ISO/IEC directives.

In the Rationale (Informative) volume of IEEE Std. 1003.1-200x, the words *shall*, *should*, and *may* are sometimes used to illustrate similar usages in IEEE Std. 1003.1-200x. However, the rationale itself does not specify anything regarding implementations or applications.

conformance document

As a practical matter, the conformance document is effectively part of the system documentation. Conformance documents are distinguished by IEEE Std. 1003.1-200x so that they can be referred to distinctly.

implementation-defined

This definition is analogous to that of the ISO C standard and, together with *undefined* and *unspecified*, provides a range of specification of freedom allowed to the interface implementor.

may

The use of *may* has been limited as much as possible, due both to confusion stemming from its ordinary English meaning and to objections regarding the desirability of having as few options as possible and those as clearly specified as possible.

The usage of *can* and *may* were selected to contrast optional application behavior (can) against optional implementation behavior (may).

shall

Declarative sentences are sometimes used in IEEE Std. 1003.1-200x as if they included the word *shall*, and facilities thus specified are no less required. For example, the two statements:

1. The *foo*( ) function shall return zero.
2. The *foo*( ) function returns zero.

are meant to be exactly equivalent.

should

In IEEE Std. 1003.1-200x, the word *should* does not usually apply to the implementation, but rather to the application. Thus, the important words regarding implementations are *shall*, which indicates requirements, and *may*, which indicates options.

obsolescent

The term *obsolescent* means ‘‘do not use this feature in new applications’’. The obsolescence concept is not an ideal solution, but was used as a method of increasing consensus: many more objections would be heard from the user community if some of these historical features were suddenly withdrawn without the grace period obsolescence implies. The phrase ‘‘may be considered for withdrawal in future revisions’’ implies that the result of that consideration might in fact keep those features indefinitely if the predominance of applications do not migrate away from them quickly.

legacy

The term *legacy* was added for compatibility with the Single UNIX Specification. It means ‘‘this feature is historic and optional; do not use this feature in new applications. There are alternate interfaces that are more suitable.’’. It is used exclusively for XSI extensions, and includes facilities that were mandatory in previous versions of the base document but are optional in this revision. This is a way to ‘‘sunset’’ the usage of certain functions. Application writers should not rely on the existence of these facilities in new applications, but should follow the migration path detailed in the APPLICATION USAGE sections of the relevant pages.

The terms *legacy* and *obsolescent* are different: a feature marked LEGACY is not recommended for new work and need not be present on an implementation (if the XSI Legacy Option Group is not supported). A feature noted as obsolescent is supported by all implementations, but may be removed in a future revision; new applications should not use these features.

system documentation

The system documentation should normally describe the whole of the implementation, including any extensions provided by the implementation. Such documents normally contain information at least as detailed as the specifications in IEEE Std. 1003.1-200x. Few requirements are made on the system documentation, but the term is needed to avoid a dangling pointer where the conformance document is permitted to point to the system documentation.

undefined

See *implementation-defined*.

unspecified

See *implementation-defined*.

The definitions for *unspecified* and *undefined* appear nearly identical at first examination, but are not. The term *unspecified* means that a conforming program may deal with the unspecified behavior, and it should not care what the outcome is. The term *undefined* says that a conforming program should not do it because no definition is provided for what it does (and implicitly it would care what the outcome was if it tried it). It is important to remember, however, that if the syntax permits the statement at all, it must have some outcome in a real implementation.

Thus, the terms *undefined* and *unspecified* apply to the way the application should think about the feature. In terms of the implementation, it is always ‘‘defined’’—there is always some result, even if it is an error. The implementation is free to choose the behavior it prefers.

This also implies that an implementation, or another standard, could specify or define the result in a useful fashion. The terms apply to IEEE Std. 1003.1-200x specifically.

The term *implementation-defined* implies requirements for documentation that are not required for *undefined* (or *unspecified*). Where there is no need for a conforming program to know the definition, the term *undefined* is used, even though *implementation-defined* could also have been used in this context. There could be a fourth term, specifying ‘‘this standard does not say what this does; it is acceptable to define it in an implementation, but it does not need to be documented’’, and undefined would then be used very rarely for the few things for which any definition is not useful.

In many places IEEE Std. 1003.1-200x is silent about the behavior of some possible construct. For example, a variable may be defined for a specified range of values and behaviors are described for those values; nothing is said about what happens if the variable has any other value. That kind of silence can imply an error in the standard, but it may also imply that the standard was intentionally silent and that any behavior is permitted. There is a natural tendency to infer that if the standard is silent, a behavior is prohibited. That is not the intent. Silence is intended to be equivalent to the term *unspecified*.

The term *application* is not defined in IEEE Std. 1003.1-200x; it is assumed to be a part of general computer science terminology.

A.1.5 Portability

To aid the identification of options within IEEE Std. 1003.1-200x, a notation consisting of margin codes and shading is used. This is based on the notation used in previous revisions of The Open Group's Base specifications.

The benefits of this approach is a reduction in the number of *if* statements within the running text, that makes the text easier to read, and also an identification to the programmer that they need to ensure that their target platforms support the underlying options. For example, if functionality is marked with THR in the margin, it will be available on all systems supporting the Threads option, but may not be available on some.

A.1.5.1 Codes

The set of code includes codes for options defined in clause 2.1.6, Options, and the following additional codes for other purposes:

CX This margin code is used to denote extensions beyond the ISO C standard, and is used in interfaces that are duplicated between IEEE Std. 1003.1-200x and the ISO C standard.

MAN This margin code was used during the development of the drafts and should not be present in the final published standard.

OB This margin code is used to denote obsolescent behavior and thus flag a possible future application portability warning.

OH The Single UNIX Specification has historically tried to reduce the number of headers an application has had to include when using a particular interface. Sometimes this was fewer than the base standard, and hence a notation is used to flag which headers are optional if you are using a system supporting the XSI extension.

PI This is another code used in the XSI extension only. It is used to denote a possible application portability warning related to behavior of an interface which may not be consistent between all conformant systems.

UN This is another code used in the XSI extension only. It is used to denote a possible application portability warning related to possibly unsupportable functionality.

XSI This code is used to denote interfaces and facilities within interfaces only required on systems supporting the XSI extension. This is introduced to support the Single UNIX Specification.

XSR This code is used to denote interfaces and facilities within interfaces only required on systems supporting STREAMS. This is introduced to support the Single UNIX Specification, although it is defined in a way so that it can standalone from the XSI notation.

A.1.5.2 Margin Code Notation

Since some features may depend on one or more options, or require more than one options, a notation is used. Where a feature requires support of a single option, a single margin code will occur in the margin. If it depends on two options and both are required, then the codes will appear with a <space> separator. If either of two options are required then a logical OR is denoted using the '|' symbol. If more than two codes are used, a special margin code is used.

A.2 Conformance

The terms *profile* and *profiling* are used throughout this section.

A profile of a standard or standards is a codified set of option selections, such that by being conformant to a profile, particular classes of users are specifically supported.

These conformance definitions are descended from those in the ISO POSIX-1:1996 standard, but with changes for the following:

- The addition of profiling options, allowing both sub-profiling as per IEEE Std. 1003.13-1998, and larger profiles of options such as the XSI extension used by the Single UNIX Specification. In effect, it has profiled itself (that is, created a self-profile).
- The addition of a hierarchy of super-options for XSI; these were formerly known as *Feature Groups* in The Open Group System Interfaces and Headers, Issue 5 specification.
- Options from the ISO POSIX-2:1993 standard are also now included as IEEE Std. 1003.1-200x merges the functionality from it.

A.2.1 Implementation Conformance

These definitions allow application developers to know what to depend on in an implementation.

There is no definition of a *strictly conforming implementation*; that would be an implementation that provides *only* those facilities specified by POSIX.1 with no extensions whatsoever. This is because no actual operating system implementation can exist without system administration and initialization facilities that are beyond the scope of POSIX.1.

A.2.1.1 Requirements

The word ''support'' is used in certain instances, rather than ''provide'', in order to allow an implementation that has no resident software development facilities, but that supports the execution of a *Strictly Conforming POSIX.1 Application*, to be a *conforming implementation*.

A.2.1.2 Documentation

The conformance documentation is required to use the same numbering scheme as POSIX.1 for purposes of cross-referencing. All options that an implementation chooses shall be reflected in **<limits.h>** and **<unistd.h>**.

Note that the use of ''may'' in terms of where conformance documents record where implementations may vary, implies that it is not required to describe those features identified as undefined or unspecified.

Other aspects of systems must be evaluated by purchasers for suitability. Many systems incorporate buffering facilities, maintaining updated data in volatile storage and transferring such updates to non-volatile storage asynchronously. Various exception conditions, such as a power failure or a system crash, can cause this data to be lost. The data may be associated with a file that is still open, with one that has been closed, with a directory, or with any other internal system data structures associated with permanent storage. This data can be lost, in whole or part, so that only careful inspection of file contents could determine that an update did not occur.

Also, interrelated file activities, where multiple files and/or directories are updated, or where space is allocated or released in the file system structures, can leave inconsistencies in the relationship between data in the various files and directories, or in the file system itself. Such inconsistencies can break applications that expect updates to occur in a specific sequence, so that updates in one place correspond with related updates in another place.

For example, if a user creates a file, places information in the file, and then records this action in another file, a system or power failure at this point followed by restart may result in a state in which the record of the action is permanently recorded, but the file created (or some of its information) has been lost. The consequences of this to the user may be undesirable. For a user on such a system, the only safe action may be to require the system administrator to have a policy that requires, after any system or power failure, that the entire file system must be restored from the most recent backup copy (causing all intervening work to be lost).

The characteristics of each implementation will vary in this respect and may or may not meet the requirements of a given application or user. Enforcement of such requirements is beyond the scope of POSIX.1. It is up to the purchaser to determine what facilities are provided in an implementation that affect the exposure to possible data or sequence loss, and also what underlying implementation techniques and/or facilities are provided that reduce or limit such loss or its consequences.

A.2.1.3 POSIX Conformance

This really means conformance to the base standard; however, since this revision includes the core material of the Single UNIX Specification, the standard developers decided that it was appropriate to segment the conformance requirements into two, the former for the base standard, and the latter for the Single UNIX Specification.

Within POSIX.1 there are some symbolic constants that, if defined, indicate that a certain option is enabled. Other symbolic constants exist in POSIX.1 for other reasons.

To enable support for sub-profiling the following options were defined:

- _POSIX_C_LANG_SUPPORT
- _POSIX_DEVICE_IO
- _POSIX_DEVICE_SPECIFIC
- _POSIX_FD_MGMT
- _POSIX_FIFO
- _POSIX_FILE_ATTRIBUTES
- _POSIX_FILE_SYSTEM
- _POSIX_MULTIPLE_PROCESS
- _POSIX_PIPE
- _POSIX_SIGNALS
- _POSIX_SINGLE_PROCESS
- _POSIX_SYSTEM_DATABASE
- _POSIX_USER_GROUPS
- _POSIX_NETWORKING

These are all mandatory in the base standard.

As part of the revision some alignment has occurred of the options with the FIPS 151-2 profile on the POSIX.1-1990 standard. The following options from the POSIX.1-1990 standard are now mandatory:

- _POSIX_JOB_CONTROL
- _POSIX_SAVED_IDS
- _POSIX_VDISABLE

A POSIX-conformant system may support the XSI extensions of the Single UNIX Specification. This was intentional since the standard developers intend them to be upwards-compatible, so that a system conforming to the Single UNIX Specification can also conform to the base standard at the same time.

A.2.1.4 XSI Conformance

This section is added since the revision merges in the base volumes of the Single UNIX Specification.

XSI conformance can be thought of as a profile, selecting certain options from IEEE Std. 1003.1-200x.

A.2.1.5 Option Groups

The concept of *Option Groups* is introduced to IEEE Std. 1003.1-200x to allow collections of related functions or options to be grouped together. This is used in two ways in IEEE Std. 1003.1-200x:

1. Firstly, for profiling, a set of *Profiling Option Groups* has been created to support subsetting of the system interfaces provided in IEEE Std. 1003.1-200x. The subsets used by IEEE Std. 1003.13-1998 were used as an initial model for those created.

2. Secondly, the *XSI Option Groups* have been created to allow super-options, collections of underlying options and related functions, to be collectively supported by XSI-conforming systems. These reflect the *Feature Groups* from The Open Group System Interfaces and Headers, Issue 5 specification.

A.2.1.6 Options

The final subsections within *Implementation Conformance* list the core options within IEEE Std. 1003.1-200x. This includes both options for the System Interfaces volume of IEEE Std. 1003.1-200x and the Shell and Utilities volume of IEEE Std. 1003.1-200x.

A.2.2 Application Conformance

These definitions guide users or adaptors of applications in determining on which implementations an application will run and how much adaptation would be required to make it run on others. These definitions are modeled after related ones in the the ISO C standard.

POSIX.1 occasionally uses the expressions *portable application* or *conforming application* . As they are used, these are synonyms for any of these three terms. The differences between the classes of application conformance relate to the requirements for other standards, the options supported (such as the XSI extension) or, in the case of the Conforming POSIX.1 Application Using Extensions, to implementation extensions. When one of the less explicit expressions is used, it should be apparent from the context of the discussion which of the more explicit names is appropriate

A.2.2.1 Strictly Conforming POSIX Application

This definition is analogous to that of a ISO C standard *conforming program*.

The major difference between a *Strictly Conforming POSIX Application* and a ISO C standard *strictly conforming program* is that the latter is not allowed to use features of POSIX that are not in the ISO C standard.

A.2.2.2 Conforming POSIX Application

Examples of <National Bodies> include ANSI, BSI, and AFNOR.

A.2.2.3 Conforming POSIX Application Using Extensions

Due to possible requirements for configuration or implementation characteristics in excess of the specifications in **<limits.h>** or related to the hardware (such as array size or file space), not every Conforming POSIX Application Using Extensions will run on every conforming implementation.

A.2.2.4 Strictly Conforming XSI Application

This is intended to be upwards-compatible with the definition of a Strictly Conforming POSIX Application, with the addition of the facilities and functionality included in the XSI extension.

A.2.2.5 Conforming XSI Application Using Extensions

Such applications may use extensions beyond the facilities defined by IEEE Std. 1003.1-200x including the XSI extension, but need to document the additional requirements.

A.2.3 Language-Dependent Services for the C Programming Language

POSIX.1 is, for historical reasons, both a specification of an operating system interface, shell and utilities, and a C binding for that specification. Efforts had been previously undertaken to generate a language-independent specification; however, that had failed, and the fact that the ISO C standard is the *de facto* primary language on POSIX and the UNIX system makes this a necessary and workable situation.

A.3 Definitions

The definitions in this section are stated so that they can be used as exact substitutes for the terms in text. They should not contain requirements or cross-references to sections within IEEE Std. 1003.1-200x; that is accomplished by using an informative note. In addition, the term should not be included in its own definition. Where requirements or descriptions need to be addressed but cannot be included in the definitions, due to not meeting the above criteria, these occur in the General Concepts chapter.

Many of these definitions are necessarily circular, and some of the terms (such as *process*) are variants of basic computing science terms that are inherently hard to define. Where some definitions are more conceptual and contain requirements, these appear in the General Concepts chapter. Those listed in this section appear in an alphabetical glossary format of terms.

Some definitions must allow extension to cover terms or facilities that are not explicitly mentioned in IEEE Std. 1003.1-200x. For example, the definition of *Extended Security Controls* permits implementations beyond those defined in IEEE Std. 1003.1-200x.

Some terms in the following list of notes do not appear in POSIX.1; these are marked prefixed with a asterisk (*). Many of them have been specifically excluded from POSIX.1 because they concern system administration, implementation, or other issues that are not specific to the programming interface. Those are marked with a reason, such as ‘‘implementation-defined’’.

Appropriate Privileges

One of the fundamental security problems with many historical UNIX systems has been that the privilege mechanism is monolithic—a user has either no privileges or *all* privileges. Thus, a successful ‘‘trojan horse’’ attack on a privileged process defeats all security provisions. Therefore, POSIX.1 allows more granular privilege mechanisms to be defined. For many historical implementations of the UNIX system, the presence of the term *appropriate privileges* in POSIX.1 may be understood as a synonym for *superuser* (UID 0). However, other systems have emerged where this is not the case and each discrete controllable action has *appropriate privileges* associated with it. Because this mechanism is implementation-defined, it must be described in the conformance document. Although that description affects several parts of POSIX.1 where the term *appropriate privilege* is used, because the term *implementation-defined* only appears here, the description of the entire mechanism and its effects on these other sections belongs in this equivalent section of the conformance document. This is especially convenient for implementations with a single mechanism that applies in all areas, since it only needs to be described once.

Character

The term *character* is used to mean a sequence of one or more bytes representing a single graphic symbol. The deviation in the exact text of the ISO C standard definition for *byte* meets the intent of the rationale of the ISO C standard also clears up the ambiguity raised by the term *basic execution character set*. The octet-minimum requirement is a reflection of the {CHAR_BIT} value.

Clock Tick

The ISO C standard defines a similar interval for use by the *clock*( ) function. There is no requirement that these intervals be the same. In historical implementations these intervals are different.

Command

The terms *command* and *utility* are related but have distinct meanings. to perform a specific task. The directive can be in the form of a single utility name (for example, *ls*), or the directive can take the form of a compound command (for example, `"ls | grep name | pr"`). A utility is a program that can be called by name from a shell. Issuing only the name of the utility to a shell is the equivalent of a one-word command. A utility may be invoked as a separate program that executes in a different process than the command language interpreter, or it may be implemented as a part of the command language interpreter. For example, the *echo* command (the directive to perform a specific task) may be implemented such that the *echo* utility (the logic that performs the task of echoing) is in a separate program; therefore, it is executed in a process that is different than the command language interpreter. Conversely, the logic that performs the *echo* utility could be built into the command language interpreter; therefore, it could execute in the same process as the command language interpreter.

The terms *tool* and *application* can be thought of as being synonymous with *utility* from the perspective of the operating system kernel. Tools, applications, and utilities historically have run, typically, in processes above the kernel level. Tools and utilities historically have been a part of the operating system non-kernel code and have performed system-related functions, such as listing directory contents, checking file systems, repairing file systems, or extracting system status information. Applications have not generally been a part of the operating system, and they perform non-system-related functions, such as word processing, architectural design, mechanical design, workstation publishing, or financial analysis. Utilities have most frequently been provided by the operating system distributor, applications by third-party software distributors, or by the users themselves. Nevertheless, IEEE Std. 1003.1-200x does not differentiate between tools, utilities, and applications when it comes to receiving services from the system, a shell, or the standard utilities. (For example, the *xargs* utility invokes another utility; it would be of fairly limited usefulness if the users could not run their own applications in place of the standard utilities.) Utilities are not applications in the sense that they are not themselves subject to the restrictions of IEEE Std. 1003.1-200x or any other standard—there is no requirement for *grep*, *stty*, or any of the utilities defined here to be any of the classes of conforming applications.

Column Positions

In most 1-bit character sets, such as ASCII, the concept of column positions is identical to character positions and to bytes. Therefore, it has been historically acceptable for some implementations to describe line folding or tab stops or table column alignment in terms of bytes or character positions. Other character sets pose complications, as they can have internal representations longer than one octet and they can have display characters that have different widths on the terminal screen or printer.

In IEEE Std. 1003.1-200x the term *column positions* has been defined to mean character—not byte—positions in input files (such as ‘‘column position 7 of the FORTRAN input’’). Output files describe the column position in terms of the display width of the narrowest printable character in the character set, adjusted to fit the characteristics of the output device. It is very possible that *n* column positions will not be able to hold *n* characters in some character sets, unless all of those characters are of the narrowest width. It is assumed that the implementation is aware of the width of the various characters, deriving this information from the value of *LC_CTYPE*, and thus can determine how many column positions to allot for each character in those utilities where it is important.

The term *column position* was used instead of the more natural *column* because the latter is frequently used in the different contexts of columns of figures, columns of table values, and so on. Wherever confusion might result, these latter types of columns are referred to as *text columns*.

Controlling Terminal

The question of which of possibly several special files referring to the terminal is meant is not addressed in POSIX.1. The file name **/dev/tty** is a synonym for the controlling terminal associated with a process.

Device Number*

The concept is handled in *stat*( ) as *ID of device*.

Direct I/O

Historically, direct I/O refers to the system bypassing intermediate buffering, but may be extended to cover implementation-defined optimizations.

Directory

The format of the directory file is implementation-defined and differs radically between System V and 4.3 BSD. However, routines (derived from 4.3 BSD) for accessing directories and certain constraints on the format of the information returned by those routines are described in the **<dirent.h>** header.

Directory Entry

Throughout IEEE Std. 1003.1-200x, the term *link* is used (about the *link*( ) function, for example) in describing the objects that point to files from directories.

Display

The Shell and Utilities volume of IEEE Std. 1003.1-200x assigns precise requirements for the terms *display* and *write*. Some historical systems have chosen to implement certain utilities without using the traditional file descriptor model. For example, the *vi* editor might employ direct screen memory updates on a personal computer, rather than a *write*( ) system call. An instance of user prompting might appear in a dialog box, rather than with standard error. When the Shell and Utilities volume of IEEE Std. 1003.1-200x uses the term *display*, the method of outputting to the terminal is unspecified; many historical implementations use *termcap* or *terminfo*, but this is not a requirement. The term *write* is used when the Shell and Utilities volume of IEEE Std. 1003.1-200x mandates that a file descriptor be used and that the output can be redirected. However, it is assumed that when the writing is directly to the terminal (it has not been redirected elsewhere), there is no practical way for a user or test suite to determine whether a file descriptor is being used. Therefore, the use of a file descriptor is mandated only for the redirection case and the implementation is free to use any method when the output is not redirected. The verb *write* is used almost exclusively, with the very few exceptions of those utilities where output redirection need not be supported: *tabs*, *talk*, *tput*, and *vi*.

Dot

The symbolic name *dot* is carefully used in POSIX.1 to distinguish the working directory file name from a period or a decimal point.

Dot-Dot

Historical implementations permit the use of these file names without their special meanings. Such use precludes any meaningful use of these file names by a Conforming POSIX.1 Application. Therefore, such use is considered an extension, the use of which makes an implementation non-conforming; see also Section A.4.9 (on page 3346).

Epoch

Historically, the origin of UNIX system time was referred to as ‘‘00:00:00 GMT, January 1, 1970’’. Greenwich Mean Time is actually not a term acknowledged by the international standards community; therefore, this term, *Epoch*, is used to abbreviate the reference to the actual standard, Coordinated Universal Time.

FIFO Special File

See *pipe* in **Pipe** (on page 3331).

File

It is permissible for an implementation-defined file type to be non-readable or non-writable.

File Classes

These classes correspond to the historical sets of permission bits. The classes are general to allow implementations flexibility in expanding the access mechanism for more stringent security environments. Note that a process is in one and only one class, so there is no ambiguity.

File Name

At the present time, the primary responsibility for truncating file names containing multi-byte characters must reside with the application. Some industry groups involved in internationalization believe that in the future the responsibility must reside with the kernel. For the moment, a clearer understanding of the implications of making the kernel responsible for truncation of multi-byte file names is needed.

Character-level truncation was not adopted because there is no support in POSIX.1 that advises how the kernel distinguishes between single and multi-byte characters. Until that time, it must be incumbent upon application writers to determine where multi-byte characters must be truncated.

File System

Historically, the meaning of this term has been overloaded with two meanings: that of the complete file hierarchy, and that of a mountable subset of that hierarchy; that is, a mounted file system. POSIX.1 uses the term *file system* in the second sense, except that it is limited to the scope of a process (and a process’ root directory). This usage also clarifies the domain in which a file serial number is unique.

Graphic Character

This definition is made available for those definitions (in particular, *TZ*) which must exclude control characters.

Group Database

See **User Database** (on page 3340).

Group File*

Implementation-defined; see **User Database** (on page 3340).

Historical Implementations*

This refers to previously existing implementations of programming interfaces and operating systems that are related to the interface specified by POSIX.1.

Hosted Implementation*

This refers to a POSIX.1 implementation that is accomplished through interfaces from the POSIX.1 services to some alternate form of operating system kernel services. Note that the line between a hosted implementation and a native implementation is blurred, since most implementations will provide some services directly from the kernel and others through some indirect path. (For example, *fopen*( ) might use *open*( ); or *mkfifo*( ) might use *mknod*( ).) There is no necessary relationship between the type of implementation and its correctness, performance, and/or reliability.

Implementation*

This term is generally used instead of its synonym, *system*, to emphasize the consequences of decisions to be made by system implementors. Perhaps if no options or extensions to POSIX.1 were allowed, this usage would not have occurred.

The term *specific implementation* is sometimes used as a synonym for *implementation*. This should not be interpreted too narrowly; both terms can represent a relatively broad group of systems. For example, a hardware vendor could market a very wide selection of systems that all used the same instruction set, with some systems desktop models and others large multi-user minicomputers. This wide range would probably share a common POSIX.1 operating system, allowing an application compiled for one to be used on any of the others; this is a [*specific*]*implementation*.

However, that wide range of machines probably has some differences between the models. Some may have different clock rates, different file systems, different resource limits, different network connections, and so on, depending on their sizes or intended usages. Even on two identical machines, the system administrators may configure them differently. Each of these different systems is known by the term *a specific instance of a specific implementation*. This term is only used in the portions of POSIX.1 dealing with runtime queries: *sysconf*( ) and *pathconf*( ).

Incomplete Path Name*

Absolute path name has been adequately defined.

Job Control

In order to understand the job control facilities in POSIX.1 it is useful to understand how they are used by a job control-cognizant shell to create the user interface effect of job control.

While the job control facilities supplied by POSIX.1 can, in theory, support different types of interactive job control interfaces supplied by different types of shells, there is historically one particular interface that is most common (provided by BSD C Shell). This discussion describes that interface as a means of illustrating how the POSIX.1 job control facilities can be used.

Job control allows users to selectively stop (suspend) the execution of processes and continue (resume) their execution at a later point. The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter (shell).

The user can launch jobs (command pipelines) in either the foreground or background. When launched in the foreground, the shell waits for the job to complete before prompting for additional commands. When launched in the background, the shell does not wait, but immediately prompts for new commands.

If the user launches a job in the foreground and subsequently regrets this, the user can type the suspend character (typically set to <control>-Z), which causes the foreground job to stop and the shell to begin prompting for new commands. The stopped job can be continued by the user (via special shell commands) either as a foreground job or as a background job. Background jobs can also be moved into the foreground via shell commands.

If a background job attempts to access the login terminal (controlling terminal), it is stopped by the terminal driver and the shell is notified, which, in turn, notifies the user. (Terminal access includes *read*( ) and certain terminal control functions, and conditionally includes *write*( ).) The user can continue the stopped job in the foreground, thus allowing the terminal access to succeed in an orderly fashion. After the terminal access succeeds, the user can optionally move the job into the background via the suspend character and shell commands.

Implementing Job Control Shells

The interactive interface described previously can be accomplished using the POSIX.1 job control facilities in the following way.

The key feature necessary to provide job control is a way to group processes into jobs. This grouping is necessary in order to direct signals to a single job and also to identify which job is in the foreground. (There is at most one job that is in the foreground on any controlling terminal at a time.)

The concept of *process groups* is used to provide this grouping. The shell places each job in a separate process group via the *setpgid*( ) function. To do this, the *setpgid*( ) function is invoked by the shell for each process in the job. It is actually useful to invoke *setpgid*( ) twice for each process: once in the child process, after calling *fork*( ) to create the process, but before calling one of the *exec* family of functions to begin execution of the program, and once in the parent shell process, after calling *fork*( ) to create the child. The redundant invocation avoids a race condition by ensuring that the child process is placed into the new process group before either the parent or the child relies on this being the case. The *process group ID* for the job is selected by the shell to be equal to the *process ID* of one of the processes in the job. Some shells choose to make one process in the job be the parent of the other processes in the job (if any). Other shells (for example, the C Shell) choose to make themselves the parent of all processes in the pipeline (job).

In order to support this latter case, the *setpgid*( ) function accepts a process group ID parameter since the correct process group ID cannot be inherited from the shell. The shell itself is considered to be a job and is the sole process in its own process group.

The shell also controls which job is currently in the foreground. A foreground and background job differ in two ways: the shell waits for a foreground command to complete (or stop) before continuing to read new commands, and the terminal I/O driver inhibits terminal access by background jobs (causing the processes to stop). Thus, the shell must work cooperatively with the terminal I/O driver and have a common understanding of which job is currently in the foreground. It is the user who decides which command should be currently in the foreground, and the user informs the shell via shell commands. The shell, in turn, informs the terminal I/O driver via the *tcsetpgrp*( ) function. This indicates to the terminal I/O driver the process group ID of the foreground process group (job). When the current foreground job either stops or terminates, the shell places itself in the foreground via *tcsetpgrp*( ) before prompting for additional commands. Note that when a job is created the new process group begins as a background process group. It requires an explicit act of the shell via *tcsetpgrp*( ) to move a process group (job) into the foreground.

When a process in a job stops or terminates, its parent (for example, the shell) receives synchronous notification by calling the *waitpid*( ) function with the WUNTRACED flag set. Asynchronous notification is also provided when the parent establishes a signal handler for SIGCHLD and does not specify the SA_NOCLDSTOP flag. Usually all processes in a job stop as a unit since the terminal I/O driver always sends job control stop signals to all processes in the process group.

To continue a stopped job, the shell sends the SIGCONT signal to the process group of the job. In addition, if the job is being continued in the foreground, the shell invokes *tcsetpgrp*( ) to place the job in the foreground before sending SIGCONT. Otherwise, the shell leaves itself in the foreground and reads additional commands.

There is additional flexibility in the POSIX.1 job control facilities that allows deviations from the typical interface. Clearing the TOSTOP terminal flag allows background jobs to perform *write*( ) functions without stopping. The same effect can be achieved on a per-process basis by having a process set the signal action for SIGTTOU to SIG_IGN.

Note that the terms *job* and *process group* can be used interchangeably. A login session that is not using the job control facilities can be thought of as a large collection of processes that are all in the same job (process group). Such a login session may have a partial distinction between foreground and background processes; that is, the shell may choose to wait for some processes before continuing to read new commands and may not wait for other processes. However, the terminal I/O driver will consider all these processes to be in the foreground since they are all members of the same process group.

In addition to the basic job control operations already mentioned, a job control-cognizant shell needs to perform the following actions.

When a foreground (not background) job stops, the shell must sample and remember the current terminal settings so that it can restore them later when it continues the stopped job in the foreground (via the *tcgetattr*( ) and *tcsetattr*( ) functions).

Because a shell itself can be spawned from a shell, it must take special action to ensure that subshells interact well with their parent shells.

A subshell can be spawned to perform an interactive function (prompting the terminal for commands) or a non-interactive function (reading commands from a file). When operating non-interactively, the job control shell will refrain from performing the job control-specific actions described above. It will behave as a shell that does not support job control. For example, all *jobs* will be left in the same process group as the shell, which itself remains in the process group established for it by its parent. This allows the shell and its children to be treated as a single job by a parent shell, and they can be affected as a unit by terminal keyboard signals.

An interactive subshell can be spawned from another job control-cognizant shell in either the foreground or background. (For example, from the C Shell, the user can execute the command, csh *&*.) Before the subshell activates job control by calling *setpgid*( ) to place itself in its own process group and *tcsetpgrp*( ) to place its new process group in the foreground, it needs to ensure that it has already been placed in the foreground by its parent. (Otherwise, there could be multiple job control shells that simultaneously attempt to control mediation of the terminal.) To determine this, the shell retrieves its own process group via *getpgrp*( ) and the process group of the current foreground job via *tcgetpgrp*( ). If these are not equal, the shell sends SIGTTIN to its own process group, causing itself to stop. When continued later by its parent, the shell repeats the process group check. When the process groups finally match, the shell is in the foreground and it can proceed to take control. After this point, the shell ignores all the job control stop signals so that it does not inadvertently stop itself.

Implementing Job Control Applications

Most applications do not need to be aware of job control signals and operations; the intuitively correct behavior happens by default. However, sometimes an application can inadvertently interfere with normal job control processing, or an application may choose to overtly effect job control in cooperation with normal shell procedures.

An application can inadvertently subvert job control processing by ''blindly'' altering the handling of signals. A common application error is to learn how many signals the system supports and to ignore or catch them all. Such an application makes the assumption that it does not know what this signal is, but knows the right handling action for it. The system may initialize the handling of job control stop signals so that they are being ignored. This allows shells that do not support job control to inherit and propagate these settings and hence to be immune to stop signals. A job control shell will set the handling to the default action and propagate this, allowing processes to stop. In doing so, the job control shell is taking responsibility for restarting the stopped applications. If an application wishes to catch the stop signals itself, it should first determine their inherited handling states. If a stop signal is being ignored, the application should continue to ignore it. This is directly analogous to the recommended handling of SIGINT described in the referenced UNIX Programmer's Manual.

If an application is reading the terminal and has disabled the interpretation of special characters (by clearing the ISIG flag), the terminal I/O driver will not send SIGTSTP when the suspend character is typed. Such an application can simulate the effect of the suspend character by recognizing it and sending SIGTSTP to its process group as the terminal driver would have done. Note that the signal is sent to the process group, not just to the application itself; this ensures that other processes in the job also stop. (Note also that other processes in the job could be children, siblings, or even ancestors.) Applications should not assume that the suspend character is <control>-Z (or any particular value); they should retrieve the current setting at startup.

Implementing Job Control Systems

The intent in adding 4.2 BSD-style job control functionality was to adopt the necessary 4.2 BSD programmatic interface with only minimal changes to resolve syntactic or semantic conflicts with System V or to close recognized security holes. The goal was to maximize the ease of providing both conforming implementations and Conforming POSIX.1 Applications.

It is only useful for a process to be affected by job control signals if it is the descendant of a job control shell. Otherwise, there will be nothing that continues the stopped process.

POSIX.1 does not specify how controlling terminal access is affected by a user logging out (that is, by a controlling process terminating). 4.2 BSD uses the *vhangup*( ) function to prevent any access to the controlling terminal through file descriptors opened prior to logout. System V does not prevent controlling terminal access through file descriptors opened prior to logout (except for the case of the special file, **/dev/tty**). Some implementations choose to make processes immune from job control after logout (that is, such processes are always treated as if in the foreground); other implementations continue to enforce foreground/background checks after logout. Therefore, a Conforming POSIX.1 Application should not attempt to access the controlling terminal after logout since such access is unreliable. If an implementation chooses to deny access to a controlling terminal after its controlling process exits, POSIX.1 requires a certain type of behavior (see **Controlling Terminal** (on page 3323)).

Kernel*

See *system call*.

Library Routine*

See *system call*.

Logical Device*

Implementation-defined.

Map

The definition of map is included to clarify the usage of mapped pages in the description of the behavior of process memory locking.

Memory-Resident

The term *memory-resident* is historically understood to mean that the so-called resident pages are actually present in the physical memory of the computer system and are immune from swapping, paging, copy-on-write faults, and so on. This is the actual intent of IEEE Std. 1003.1-200x in the process memory locking section for implementations where this is logical. But for some implementations—primarily mainframes—actually locking pages into primary storage is not advantageous to other system objectives, such as maximizing throughput. For such implementations, memory locking is a ‘‘hint’’ to the implementation that the application wishes to avoid situations that would cause long latencies in accessing memory. Furthermore, there are other implementation-defined issues with minimizing memory access latencies that ‘‘memory residency’’ does not address—such as MMU reload faults. The definition attempts to accommodate various implementations while allowing portable applications to specify to the implementation that they want or need the best memory access times that the implementation can provide.

Memory Object*

The term *memory object* usually implies shared memory. If the object is the same as a file name in the file system name space of the implementation, it is expected that the data written into the memory object be preserved on disk. A memory object may also apply to a physical device on an implementation. In this case, writes to the memory object are sent to the controller for the device and reads result in control registers being returned.

Mount Point*

The directory on which a *mounted file system* is mounted. This term, like *mount*( ) and *umount*( ), was not included because it was implementation-defined.

Mounted File System*

See *file system*.

name

There are no explicit limits in IEEE Std. 1003.1-200x on the sizes of names, words (see the definition of word in the Base Definitions volume of IEEE Std. 1003.1-200x), lines, or other objects. However, other implicit limits do apply: shell script lines produced by many of the standard utilities cannot exceed {LINE_MAX} and the sum of exported variables comes under the {ARG_MAX} limit. Historical shells dynamically allocate memory for names and words and parse incoming lines a byte at a time. Lines cannot have an arbitrary {LINE_MAX} limit because of historical practice, such as makefiles, where *make* removes the <newline> characters associated with the commands for a target and presents the shell with one very long line. The text on INPUT FILES in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 1.11, Utility Description Defaults does allow a shell to run out of memory, but it cannot have arbitrary programming limits.

Native Implementation*

This refers to an implementation of POSIX.1 that interfaces directly to an operating system kernel; see also *hosted implementation* and *cooperating implementation*. A similar concept is a native UNIX system, which would be a kernel derived from one of the original UNIX system products.

Nice Value

This definition is not intended to suggest that all processes in a system have priorities that are comparable. Scheduling policy extensions, such as adding realtime priorities, make the notion of a single underlying priority for all scheduling policies problematic. Some systems may implement the features related to *nice* to affect all processes on the system, others to affect just the general time-sharing activities implied by IEEE Std. 1003.1-200x, and others may have no effect at all. Because of the use of ''implementation-defined'' in *nice* and *renice*, a wide range of implementation strategies is possible.

Open File Description

An *open file description*, as it is currently named, describes how a file is being accessed. What is currently called a *file descriptor* is actually just an identifier or ''handle''; it does not actually describe anything.

The following alternate names were discussed:

- For *open file description*:
  open instance, *file access description*, *open file information*, and *file access information*.
- For *file descriptor*:
  file handle, *file number* (c.f., *fileno*( )). Some historical implementations use the term *file table entry*.

Orphaned Process Group

Historical implementations have a concept of an orphaned process, which is a process whose parent process has exited. When job control is in use, it is necessary to prevent processes from being stopped in response to interactions with the terminal after they no longer are controlled by a job control-cognizant program. Because signals generated by the terminal are sent to a process group and not to individual processes, and because a signal may be provoked by a process that is not orphaned, but sent to another process that is orphaned, it is necessary to define an orphaned process group. The definition assumes that a process group will be manipulated as a group and that the job control-cognizant process controlling the group is outside of the group and is the parent of at least one process in the group (so that state changes may be reported via *waitpid*( )). Therefore, a group is considered to be controlled as long as at least one process in the group has a parent that is outside of the process group, but within the session.

This definition of orphaned process groups ensures that a session leader's process group is always considered to be orphaned, and thus it is prevented from stopping in response to terminal signals.

Page

The term *page* is defined to support the description of the behavior of memory mapping for shared memory and memory mapped files, and the description of the behavior of process memory locking. It is not intended to imply that shared memory/file mapping and memory locking are applicable only to ''paged'' architectures. For the purposes of IEEE Std. 1003.1-200x, whatever the granularity on which an architecture supports mapping or locking is considered to be a ''page'' . If an architecture cannot support the memory mapping or locking functions specified by IEEE Std. 1003.1-200x on any granularity, then these options will not be implemented on the architecture.

Passwd File*

Implementation-defined; see **User Database** (on page 3340).

Parent Directory

There may be more than one directory entry pointing to a given directory in some implementations. The wording here identifies that exactly one of those is the parent directory. In *path name resolution*, dot-dot is identified as the way that the unique directory is identified. (That is, the parent directory is the one to which dot-dot points.) In the case of a remote file system, if the same file system is mounted several times, it would appear as if they were distinct file systems (with interesting synchronization properties).

Pipe

It proved convenient to define a pipe as a special case of a FIFO, even though historically the latter was not introduced until System III and does not exist at all in 4.3 BSD.

Portable File Name Character Set

The encoding of this character set is not specified—specifically, ASCII is not required. But the implementation must provide a unique character code for each of the printable graphics specified by POSIX.1; see also Section A.4.5 (on page 3342).

Situations where characters beyond the portable file name character set (or historically ASCII or the ISO/IEC 646:1991 standard) would be used (in a context where the portable file name character set or the ISO/IEC 646:1991 standard is required by POSIX.1) are expected to be common. Although such a situation renders the use technically non-compliant, mutual agreement among the users of an extended character set will make such use portable between those users. Such a mutual agreement could be formalized as an optional extension to POSIX.1. (Making it required would eliminate too many possible systems, as even those systems using the ISO/IEC 646:1991 standard as a base character set extend their character sets for Western Europe and the rest of the world in different ways.)

Nothing in POSIX.1 is intended to preclude the use of extended characters where interchange is not required or where mutual agreement is obtained. It has been suggested that in several places ‘‘should’’ be used instead of ‘‘shall’’. Because (in the worst case) use of any character beyond the portable file name character set would render the program or data not portable to all possible systems, no extensions are permitted in this context.

Regular File

POSIX.1 does not intend to preclude the addition of structuring data (for example, record lengths) in the file, as long as such data is not visible to an application that uses the features described in POSIX.1.

Root Directory

This definition permits the operation of *chroot*( ), even though that function is not in POSIX.1; see also *file hierarchy*.

Root File System*

Implementation-defined.

Root of a File System*

Implementation-defined; see *mount point*.

Seconds Since the Epoch

Coordinated Universal Time uses the concept of leap seconds; at the time POSIX.1 was published, 14 leap seconds had been added since January 1, 1970. These 14 seconds are ignored to provide an easy and compatible method of computing time differences.

Most systems' notion of ‘‘time’’ is that of a continuously increasing value, so this value should increase even during leap seconds. However, not only do most systems not keep track of leap seconds, but most systems are probably not synchronized to any standard time reference. Therefore, it is inappropriate to require that a time represented as seconds since the Epoch precisely represent the number of seconds between the referenced time and the Epoch.

It is sufficient to require that applications be allowed to treat this time as if it represented the number of seconds between the referenced time and the Epoch. It is the responsibility of the vendor of the system, and the administrator of the system, to ensure that this value represents the number of seconds between the referenced time and the Epoch as closely as necessary for the application being run on that system.

It is important that the interpretation of time names and *seconds since the Epoch* values be consistent across conforming systems; that is, it is important that all conforming systems interpret ''536 457 599 seconds since the Epoch'' as 59 seconds, 59 minutes, 23 hours 31 December 1986, regardless of the accuracy of the system's idea of the current time. The expression is given to assure a consistent interpretation, not to attempt to specify the calendar. The relationship between *tm_yday* and the day of week, day of month, and month is presumed to be specified elsewhere and is not given in POSIX.1.

Consistent interpretation of *seconds since the Epoch* can be critical to certain types of distributed applications that rely on such timestamps to synchronize events. The accrual of leap seconds in a time standard is not predictable. The number of leap seconds since the Epoch will likely increase. POSIX.1 is more concerned about the synchronization of time between applications of astronomically short duration. These concerns are expected to become more critical in the future.

Note that *tm_yday* is zero-based, not one-based, so the day number in the example above is 364. Note also that the division is an integer division (discarding remainder) as in the C language.

Note also that the meaning of *gmtime*( ), *localtime*( ), and *mktime*( ) is specified in terms of this expression. However, the ISO C standard computes *tm_yday* from *tm_mday*, *tm_mon*, and *tm_year* in *mktime*( ). Because it is stated as a (bidirectional) relationship, not a function, and because the conversion between month-day-year and day-of-year dates is presumed well known and is also a relationship, this is not a problem.

Implementations that implement **time_t** as a 32-bit integer will overflow in 2 038. POSIX.1 does not Specify the data size for **time_t**.

See also **Epoch** (on page 3324).

Signal

The definition implies a double meaning for the term. Although a signal is an event, common usage implies that a signal is an identifier of the class of event.

Superuser*

This concept, with great historical significance to UNIX system users, has been replaced with the notion of appropriate privileges.

Supplementary Group ID

The POSIX.1-1990 standard is inconsistent in its treatment of supplementary groups. The definition of supplementary group ID explicitly permits the effective group ID to be included in the set, but wording in the description of the *setuid*( ) and *setgid*( ) functions states: ''Any supplementary group IDs of the calling process remain unchanged by these function calls''. In the case of *setgid*( ) this contradicts that definition. In addition, some felt that the unspecified behavior in the definition of supplementary group IDs adds unnecessary portability problems. The standard developers considered several solutions to this problem:

1. Reword the description of *setgid*( ) to permit it to change the supplementary group IDs to reflect the new effective group ID. A problem with this is that it adds more ''may''s to the wording and does not address the portability problems of this optional behavior.

2. Mandate the inclusion of the effective group ID in the supplementary set (giving {NGROUPS_MAX} a minimum value of 1). This is the behavior of 4.4 BSD. In that system, the effective group ID is the first element of the array of supplementary group IDs (there is no separate copy stored, and changes to the effective group ID are made only in the

supplementary group set). By convention, the initial value of the effective group ID is duplicated elsewhere in the array so that the initial value is not lost when executing a set-group-ID program.

3. Change the definition of supplementary group ID to exclude the effective group ID and specify that the effective group ID does not change the set of supplementary group IDs. This is the behavior of 4.2 BSD, 4.3 BSD, and System V, Release 4.

4. Change the definition of supplementary group ID to exclude the effective group ID, and require that *getgroups*( ) return the union of the effective group ID and the supplementary group IDs.

5. Change the definition of {NGROUPS_MAX} to be one more than the number of supplementary group IDs, so it continues to be the number of values returned by *getgroups*( ) and existing applications continue to work. This alternative is effectively the same as the second (and might actually have the same implementation).

The standard developers decided to permit either 2 or 3. The effective group ID is orthogonal to the set of supplementary group IDs, and it is implementation-defined whether *getgroups*( ) returns this. If the effective group ID is returned with the set of supplementary group IDs, then all changes to the effective group ID affect the supplementary group set returned by *getgroups*( ). It is permissible to eliminate duplicates from the list returned by *getgroups*( ). However, if a group ID is contained in the set of supplementary group IDs, setting the group ID to that value and then to a different value should not remove that value from the supplementary group IDs.

The definition of supplementary group IDs has been changed to not include the effective group ID. This simplifies permanent rationale and makes the relevant functions easier to understand. The *getgroups*( ) function has been modified so that it can, on an implementation-defined basis, return the effective group ID. By making this change, functions that modify the effective group ID do not need to discuss adding to the supplementary group list; the only view into the supplementary group list that the application writer has is through the *getgroups*( ) function.

Symbolic Link

Many implementations associate no attributes, including ownership with symbolic links. Security experts encouraged consideration for defining these attributes as optional. Consideration was given to changing *utime*( ) to allow modification of the times for a symbolic link, or as an alternative adding an *lutime*( ) interface. Modifications to *chown*( ) were also considered: allow changing symbolic link ownership or alternatively adding *lchown*( ). As a result of the problems encountered in defining attributes for symbolic links (and interfaces to access/modify those attributes) and since implementations exist that do not associate these attributes with symbolic links, only the file type bits in the *st_mode* member and the *st_size* member of the **stat** structure are required to be applicable to symbolic links.

Historical implementations were followed when determining which interfaces should apply to symbolic links. Interfaces that historically followed symbolic links include *chmod*( ), *link*( ), and *utime*( ). Interfaces that historically do not follow symbolic links include *chown*( ), *lstat*( ), *readlink*( ), *rename*( ), *remove*( ), *rmdir*( ), and *unlink*( ). IEEE Std. 1003.1-200x deviates from historical practice only in the case of *chown*( ). Because there is no requirement that there be an association of ownership with symbolic links, there was no point in requiring an interface to change ownership. In addition, other implementations of symbolic links have modified *chown*( ) to follow symbolic links.

In the case of symbolic links, IEEE Std. 1003.1-200x states that a trailing slash is considered to be the final component of a path name rather than the path name component that preceded it. This is the behavior of historical implementations. For example, for **/a/b** and **/a/b/**, if **/a/b** is a symbolic link to a directory, then **/a/b** refers to the symbolic link, and **/a/b/** is the same as **/a/b/.**, which is the directory to which the symbolic link points.

For multi-level security purposes, it is possible to have the link read mode govern permission for the *readlink*( ) function. It is also possible that the read permissions of the directory containing the link be used for this purpose. Implementations may choose to use either of these methods; however, this is not current practice and neither method is specified.

Several reasons were advanced for requiring that when a symbolic link is used as the source argument to the *link*( ) function, the resulting link will apply to the file named by the contents of the symbolic link rather than to the symbolic link itself. This is the case in historical implementations. This action was preferred, as it supported the traditional idea of persistence with respect to the target of a hard link. This decision is appropriate in light of a previous decision not to require association of attributes with symbolic links, thereby allowing implementations which do not use inodes. Opposition centered on the lack of symmetry on the part of the *link*( ) and *unlink*( ) function pair with respect to symbolic links.

Because a symbolic link and its referenced object coexist in the file system name space, confusion can arise in distinguishing between the link itself and the referenced object. Historically, utilities and system calls have adopted their own link following conventions in a somewhat *ad hoc* fashion. Rules for a uniform approach are outlined here, although historical practice has been adhered to as much as was possible. To promote consistent system use, user-written utilities are encouraged to follow these same rules.

Symbolic links are handled either by operating on the link itself, or by operating on the object referenced by the link. In the latter case, an application or system call is said to follow the link. Symbolic links may reference other symbolic links, in which case links are dereferenced until an object that is not a symbolic link is found, a symbolic link that references a file that does not exist is found, or a loop is detected. (Current implementations do not detect loops, but have a limit on the number of symbolic links that they will dereference before declaring it an error.)

There are four domains for which default symbolic link policy is established in a system. In almost all cases, there are utility options that override this default behavior. The four domains are as follows:

1. Symbolic links specified to system calls that take file name arguments
2. Symbolic links specified as command line file name arguments to utilities that are not performing a traversal of a file hierarchy
3. Symbolic links referencing files not of type directory, specified to utilities that are performing a traversal of a file hierarchy
4. Symbolic links referencing files of type directory, specified to utilities that are performing a traversal of a file hierarchy

First Domain

The first domain is considered in earlier rationale.

Second Domain

The reason this category is restricted to utilities that are not traversing the file hierarchy is that some standard utilities take an option that specifies a hierarchical traversal, but by default operate on the arguments themselves. Generally, users specifying the option for a file hierarchy traversal wish to operate on a single, physical hierarchy, and therefore symbolic links, which may reference files outside of the hierarchy, are ignored. For example, *chown owner file* is a different operation from the same command with the **–R** option specified. In this example, the behavior of the command *chown owner file* is described here, while the behavior of the command *chown* **–R** *owner file* is described in the third and fourth domains.

The general rule is that the utilities in this category follow symbolic links named as arguments.

Exceptions in the second domain are:

- The *mv* and *rm* utilities do not follow symbolic links named as arguments, but respectively attempt to rename or delete them.
- The *ls* utility is also an exception to this rule. For compatibility with historical systems, when the **–R** option is not specified, the *ls* utility follows symbolic links named as arguments if the **–L** option is specified or if the **–F**, **–d**, or **–l** options are not specified. (If the **–L** option is specified, *ls* always follows symbolic links; it is the only utility where the **–L** option affects its behavior even though a tree walk is not being performed.)

All other standard utilities, when not traversing a file hierarchy, always follow symbolic links named as arguments.

Historical practice is that the **–h** option is specified if standard utilities are to act upon symbolic links instead of upon their targets. Examples of commands that have historically had a **–h** option for this purpose are the *chgrp*, *chown*, *file*, and *test* utilities.

Third Domain

The third domain is symbolic links, referencing files not of type directory, specified to utilities that are performing a traversal of a file hierarchy. (This includes symbolic links specified as command line file name arguments or encountered during the traversal.)

The intention of the Shell and Utilities volume of IEEE Std. 1003.1-200x is that the operation that the utility is performing is applied to the symbolic link itself, if that operation is applicable to symbolic links. The reason that the operation is not required is that symbolic links in some systems do not have such attributes as a file owner, and therefore the *chown* operation would be meaningless. If symbolic links on the system have an owner, it is the intention that the utility *chown* cause the owner of the symbolic link to change. If symbolic links do not have an owner, the symbolic link should be ignored. Specifically, by default, no change should be made to the file referenced by the symbolic link.

Fourth Domain

The fourth domain is symbolic links referencing files of type directory, specified to utilities that are performing a traversal of a file hierarchy. (This includes symbolic links specified as command line file name arguments or encountered during the traversal.)

All standard utilities do not, by default, indirect into the file hierarchy referenced by the symbolic link. (The Shell and Utilities volume of IEEE Std. 1003.1-200x uses the informal term *physical walk* to describe this case. The case where the utility does indirect through the symbolic link is termed a *logical walk*.)

There are three reasons for the default to a physical walk:

1. With very few exceptions, a physical walk has been the historical default on UNIX systems supporting symbolic links. Because some utilities (that is, *rm*) must default to a physical walk, regardless, changing historical practice in this regard would be confusing to users and needlessly incompatible.
2. For systems where symbolic links have the historical file attributes (that is, *owner*, *group*, *mode*), defaulting to a logical traversal would require the addition of a new option to the commands to modify the attributes of the link itself. This is painful and more complex than the alternatives.

3. There is a security issue with defaulting to a logical walk. Historically, the command *chown* –**R** *user file* has been safe for the superuser because *setuid* and *setgid* bits were lost when the ownership of the file was changed. If the walk were logical, changing ownership would no longer be safe because a user might have inserted a symbolic link pointing to any file in the tree. Again, this would necessitate the addition of an option to the commands doing hierarchy traversal to not indirect through the symbolic links, and historical scripts doing recursive walks would instantly become security problems. While this is mostly an issue for system administrators, it is preferable to not have different defaults for different classes of users.

As consistently as possible, users may cause standard utilities performing a file hierarchy traversal to follow any symbolic links named on the command line, regardless of the type of file they reference, by specifying the –**H** (for half logical) option. This option is intended to make the command line name space look like the logical name space.

As consistently as possible, users may cause standard utilities performing a file hierarchy traversal to follow any symbolic links named on the command line as well as any symbolic links encountered during the traversal, regardless of the type of file they reference, by specifying the –**L** (for logical) option. This option is intended to make the entire name space look like the logical name space.

For consistency, implementors are encouraged to use the –**P** (for physical) flag to specify the physical walk in utilities that do logical walks by default for whatever reason. The only standard utilities that require the –**P** option are *cd* and *pwd*; see the note below.

When one or more of the –**H**, –**L**, and –**P** flags can be specified, the last one specified determines the behavior of the utility. This permits users to alias commands so that the default behavior is a logical walk and then override that behavior on the command line.

Exceptions in the Third and Fourth Domains

The *ls* and *rm* utilities are exceptions to these rules. The *rm* utility never follows symbolic links and does not support the –**H**, –**L**, or –**P** options. Some historical versions of *ls* always followed symbolic links given on the command line whether the –**L** option was specified or not. Historical versions of *ls* did not support the –**H** option. In IEEE Std. 1003.1-200x, the *ls* utility never follows symbolic links unless one of the –**H** or –**L** options is specified. The *ls* utility does not support the –**P** option.

The Shell and Utilities volume of IEEE Std. 1003.1-200x requires that the standard utilities *ls*, *find*, and *pax* detect infinite loops when doing logical walks; that is, a directory, or more commonly a symbolic link, that refers to an ancestor in the current file hierarchy. If the file system itself is corrupted, causing the infinite loop, it may be impossible to recover. Because *find* and *ls* are often used in system administration and security applications, they should attempt to recover and continue as best as they can. The *pax* utility should terminate because the archive it was creating is by definition corrupted. Other, less vital, utilities should probably simply terminate as well. Implementations are strongly encouraged to detect infinite loops in all utilities.

Historical practice is shown in Table A-1 (on page 3338). The heading **SVID3** stands for the Third Edition of the System V Interface Definition.

Historically, several shells have had built-in versions of the *pwd* utility. In some of these shells, *pwd* reported the physical path, and in others, the logical path. Implementations of the shell corresponding to IEEE Std. 1003.1-200x must report the logical path by default. Earlier versions of IEEE Std. 1003.1-200x did not require the *pwd* utility to be a built-in utility. Now that *pwd* is required to set an environment variable in the current shell execution environment, it must be a built-in utility.

The *cd* command is required, by default, to treat the file name dot-dot logically. Implementors are required to support the **–P** flag in *cd* so that users can have their current environment handled physically. In 4.3 BSD, *chgrp* during tree traversal changed the group of the symbolic link, not the target. Symbolic links in 4.4 BSD do not have *owner*, *group*, *mode*, or other standard UNIX system file attributes.

Table A-1 Historical Practice for Symbolic Links

Utility	SVID3	4.3 BSD	4.4 BSD	POSIX	Comments
cd				**–L**	Treat ".." logically.
cd				**–P**	".." physically.
chgrp			**–H**	**–H**	Follow command line symlinks.
chgrp			**–h**	**–L**	Follow symlinks.
chgrp	**–h**			**–h**	Affect the symlink.
chmod				**–h**	Affect the symlink.
chmod			**–H**	**–H**	Follow command line symlinks.
chmod			**–h**	**–L**	Follow symlinks.
chown			**–H**	**–H**	Follow command line symlinks.
chown			**–h**	**–L**	Follow symlinks.
chown	**–h**			**–h**	Affect the symlink.
cp			**–H**	**–H**	Follow command line symlinks.
cp			**–h**	**–L**	Follow symlinks.
cpio	**–L**		**–L**		Follow symlinks.
du			**–H**	**–H**	Follow command line symlinks.
du			**–h**	**–L**	Follow symlinks.
file	**–h**			**–h**	Affect the symlink.
find			**–H**	**–H**	Follow command line symlinks.
find			**–h**	**–L**	Follow symlinks.
find	**–follow**		**–follow**		Follow symlinks.
ln	**–s**	**–s**	**–s**	**–s**	Create a symbolic link.
ls	**–L**	**–L**	**–L**	**–L**	Follow symlinks.
ls				**–H**	Follow command line symlinks.
mv					Operates on the symlink.
pax			**–H**	**–H**	Follow command line symlinks.
pax			**–h**	**–L**	Follow symlinks.
pwd				**–L**	Printed path may contain symlinks.
pwd				**–P**	Printed path will not contain symlinks.
rm					Operates on the symlink.
tar			**–H**		Follow command line symlinks.
tar		**–h**	**–h**		Follow symlinks.
test	**–h**		**–h**	**–h**	Affect the symlink.

Synchronously-Generated Signal

Those signals that may be generated synchronously include SIGABRT, SIGBUS, SIGILL, SIGFPE, SIGPIPE, and SIGSEGV.

System Call*

The distinction between a *system call* and a *library routine* is an implementation detail that may differ between implementations and has thus been excluded from POSIX.1.

See ''Interface, Not Implementation'' in the Introduction.

System Reboot

A *system reboot* is an event initiated by an unspecified circumstance that causes all processes (other than special system processes) to be terminated in an implementation-defined manner, after which any changes to the state and contents of files created or written to by a Conforming POSIX.1 Application prior to the event are implementation-defined.

Synchronized I/O Data (and File) Integrity Completion

These terms specify that for synchronized read operations, pending writes must be successfully completed before the read operation can complete. This is motivated by two circumstances. Firstly, when synchronizing processes can access the same file, but not share common buffers (such as for a remote file system), this requirement permits the reading process to guarantee that it can read data written remotely. Secondly, having data written synchronously is insufficient to guarantee the order with respect to a subsequent write by a reading process, and thus this extra read semantic is necessary.

Text File

The term *text file* does not prevent the inclusion of control or other non-printable characters (other than NUL). Therefore, standard utilities that list text files as inputs or outputs are either able to process the special characters or they explicitly describe their limitations within their individual descriptions. The definition of *text file* has caused controversy. The only difference between text and binary files is that text files have lines of less than {LINE_MAX} bytes, with no NUL characters, each terminated by a <newline> character. The definition allows a file with a single <newline> character, but not a totally empty file, to be called a text file. If a file ends with an incomplete line it is not strictly a text file by this definition. The <newline> character referred to in IEEE Std. 1003.1-200x is not some generic line separator, but a single character; files created on systems where they use multiple characters for ends of lines are not portable to all conforming systems without some translation process unspecified by IEEE Std. 1003.1-200x.

Thread

IEEE Std. 1003.1-200x defines a thread to be a flow of control within a process. Each thread has a minimal amount of private state; most of the state associated with a process is shared among all of the threads in the process. While most multi-thread extensions to POSIX have taken this approach, others have made different decisions.

Note: The choice to put threads within a process does not constrain implementations to implement threads in that manner. However, all functions have to behave as though threads share the indicated state information with the process from which they were created.

Threads need to share resources in order to cooperate. Memory has to be widely shared between threads in order for the threads to cooperate at a fine level of granularity. Threads keep data structures and the locks protecting those data structures in shared memory. For a data structure to be usefully shared between threads, such structures should not refer to any data that can only be interpreted meaningfully by a single thread. Thus, any system resources that might be referred to in data structures need to be shared between all threads. File descriptors, path names, and pointers to stack variables are all things that programmers want to share between their threads. Thus, the file descriptor table, the root directory, the current working directory, and the address space have to be shared.

Library implementations are possible as long as the effective behavior is as if system services invoked by one thread do not suspend other threads. This may be difficult for some library implementations on systems that do not provide asynchronous facilities.

See Section B.2.9 (on page 3447) for additional rationale.

Thread ID

See Section B.2.9.2 (on page 3463) for additional rationale.

Thread-Safe Function

All functions required by IEEE Std. 1003.1-200x need to be thread-safe; see Section A.4.14 (on page 3347) and Section B.2.9.1 (on page 3460) for additional rationale.

User Database

There are no references in IEEE Std. 1003.1-200x to a *passwd file* or a *group file*, and there is no requirement that the *group* or *passwd* databases be kept in files containing editable text. Many large timesharing systems use *passwd* databases that are hashed for speed. Certain security classifications prohibit certain information in the *passwd* database from being publicly readable.

The term *encoded* is used instead of *encrypted* in order to avoid the implementation connotations (such as reversibility or use of a particular algorithm) of the latter term.

The *getgrent*( ), *setgrent*( ), *endgrent*( ), *getpwent*( ), *setpwent*( ), and *endpwent*( ) functions are not included as part of the base standard because they provide a linear database search capability that is not generally useful (the *getpwuid*( ), *getpwnam*( ), *getgrgid*( ), and *getgrnam*( ) functions are provided for keyed lookup) and because in certain distributed systems, especially those with different authentication domains, it may not be possible or desirable to provide an application with the ability to browse the system databases indiscriminately. They are provided on XSI-conformant systems due to their historical usage by many existing applications.

A change from historical implementations is that the structures used by these functions have fields of the types **gid_t** and **uid_t**, which are required to be defined in the **<sys/types.h>** header. IEEE Std. 1003.1-200x requires implementations to ensure that these types are defined by inclusion of **<grp.h>** and **<pwd.h>**, respectively, without imposing any name space pollution or errors from redefinition of types.

IEEE Std. 1003.1-200x is silent about the content of the strings containing user or group names. These could be digit strings. IEEE Std. 1003.1-200x is also silent as to whether such digit strings bear any relationship to the corresponding (numeric) user or group ID.

Database Access

The thread-safe versions of the user and group database access functions return values in user-supplied buffers instead of possibly using static data areas that may be overwritten by each call.

Virtual Processor*

The term *virtual processor* was chosen as a neutral term describing all kernel-level scheduleable entities, such as processes, Mach tasks, or lightweight processes. Implementing threads using multiple processes as virtual processors, or implementing multiplexed threads above a virtual processor layer, should be possible, provided some mechanism has also been implemented for sharing state between processes or virtual processors. Many systems may also wish to provide implementations of threads on systems providing ‘‘shared processes’’ or ‘‘variable-weight processes’’. It was felt that exposing such implementation details would severely limit the type of systems upon which the threads interface could be supported and prevent certain types of valid implementations. It was also determined that a virtual processor interface was out of the scope of the Rationale (Informative) volume of IEEE Std. 1003.1-200x.

XSI

This is introduced to allow IEEE Std. 1003.1-200x to be adopted as an IEEE standard and an Open Group Technical Standard, serving both the POSIX and the Single UNIX Specification in a core set of volumes.

The term *XSI* has been used for 10 years in connection with the XPG series and the first and second versions of the base volumes of the Single UNIX Specification. The XSI margin code was introduced to denote the extended or more restrictive semantics beyond POSIX that are applicable to UNIX systems.

A.4 General Concepts

A.4.1 Concurrent Execution

There is no additional rationale provided for this section.

A.4.2 Extended Security Controls

Allowing an implementation to define extended security controls enables the use of IEEE Std. 1003.1-200x in environments that require different or more rigorous security than that provided in POSIX.1. Extensions are allowed in two areas: privilege and file access permissions. The semantics of these areas have been defined to permit extensions with reasonable, but not exact, compatibility with all existing practices. For example, the elimination of the superuser definition precludes identifying a process as privileged or not by virtue of its effective user ID.

A.4.3 File Access Permissions

A process should not try to anticipate the result of an attempt to access data by *a priori* use of these rules. Rather, it should make the attempt to access data and examine the return value (and possibly *errno* as well), or use *access*( ). An implementation may include other security mechanisms in addition to those specified in POSIX.1, and an access attempt may fail because of those additional mechanisms, even though it would succeed according to the rules given in this section. (For example, the user's security level might be lower than that of the object of the access attempt.) The supplementary group IDs provide another reason for a process to not attempt to anticipate the result of an access attempt.

A.4.4 File Hierarchy

Though the file hierarchy is commonly regarded to be a tree, POSIX.1 does not define it as such for three reasons:

1. Links may join branches.
2. In some network implementations, there may be no single absolute root directory; see *path name resolution*.
3. With symbolic links, the file system need not be a tree or even a directed acyclic graph.

A.4.5 File Names

Historically, certain file names have been reserved. This list includes **core**, **/etc/passwd**, and so on. Portable applications should avoid these.

Most historical implementations prohibit case folding in file names; that is, treating uppercase and lowercase alphabetic characters as identical. However, some consider case folding desirable:

- For user convenience
- For ease-of-implementation of the POSIX.1 interface as a hosted system on some popular operating systems

Variants, such as maintaining case distinctions in file names, but ignoring them in comparisons, have been suggested. Methods of allowing escaped characters of the case opposite the default have been proposed.

Many reasons have been expressed for not allowing case folding, including:

- No solid evidence has been produced as to whether case-sensitivity or case-insensitivity is more convenient for users.
- Making case-insensitivity a POSIX.1 implementation option would be worse than either having it or not having it, because:
  — More confusion would be caused among users.
  — Application developers would have to account for both cases in their code.
  — POSIX.1 implementors would still have other problems with native file systems, such as short or otherwise constrained file names or path names, and the lack of hierarchical directory structure.
- Case folding is not easily defined in many European languages, both because many of them use characters outside the USASCII alphabetic set, and because:
  — In Spanish, the digraph `"ll"` is considered to be a single letter, the capitalized form of which may be either `"Ll"` or `"LL"`, depending on context.
  — In French, the capitalized form of a letter with an accent may or may not retain the accent, depending on the country in which it is written.
  — In German, the sharp ess may be represented as a single character resembling a Greek beta (β) in lowercase, but as the digraph `"SS"` in uppercase.
  — In Greek, there are several lowercase forms of some letters; the one to use depends on its position in the word. Arabic has similar rules.
- Many East Asian languages, including Japanese, Chinese, and Korean, do not distinguish case and are sometimes encoded in character sets that use more than one byte per character.
- Multiple character codes may be used on the same machine simultaneously. There are several ISO character sets for European alphabets. In Japan, several Japanese character codes are commonly used together, sometimes even in file names; this is evidently also the case in China. To handle case insensitivity, the kernel would have to at least be able to distinguish for which character sets the concept made sense.
- The file system implementation historically deals only with bytes, not with characters, except for slash and the null byte.
- The purpose of POSIX.1 is to standardize the common, existing definition, not to change it. Mandating case-insensitivity would make all historical implementations non-standard.
- Not only the interface, but also application programs would need to change, counter to the purpose of having minimal changes to existing application code.
- At least one of the original developers of the UNIX system has expressed objection in the strongest terms to either requiring case-insensitivity or making it an option, mostly on the basis that POSIX.1 should not hinder portability of application programs across related implementations in order to allow compatibility with unrelated operating systems.

Two proposals were entertained regarding case folding in file names:

1. Remove all wording that previously permitted case folding.

   Rationale — Case folding is inconsistent with portable file name character set definition and file name definition (all characters except slash and null). No known implementations allowing all characters except slash and null also do case folding.

2. Change ''though this practice is not recommended:'' to ''although this practice is strongly discouraged.''

   Rationale If case folding must be included in POSIX.1, the wording should be stronger to discourage the practice.

The consensus selected the first proposal. Otherwise, a portable application would have to assume that case folding would occur when it was not wanted, but that it would not occur when it was wanted.

A.4.6 File Times Update

This section reflects the actions of historical implementations. The times are not updated immediately, but are only marked for update by the functions. An implementation may update these times immediately.

The accuracy of the time update values is intentionally left unspecified so that systems can control the bandwidth of a possible covert channel.

The wording was carefully chosen to make it clear that there is no requirement that the conformance document contain information that might incidentally affect file update times. Any function that performs path name resolution might update several *st_atime* fields. Functions such as *getpwnam*( ) and *getgrnam*( ) might update the *st_atime* field of some specific file or files. It is intended that these are not required to be documented in the conformance document, but they should appear in the system documentation.

A.4.7 Measurement of Execution Time

The methods used to measure the execution time of processes and threads, and the precision of these measurements, may vary considerably depending on the software architecture of the implementation, and on the underlying hardware. Implementations can also make tradeoffs between the scheduling overhead and the precision of the execution time measurements. IEEE Std. 1003.1-200x does not impose any requirement on the accuracy of the execution time; it instead specifies that the measurement mechanism and its precision are implementation-defined.

A.4.8 Memory Synchronization

In older multi-processors, access to memory by the processors was strictly multiplexed. This meant that a processor executing program code interrogates or modifies memory in the order specified by the code and that all the memory operation of all the processors in the system appear to happen in some global order, though the operation histories of different processors are interleaved arbitrarily. The memory operations of such machines are said to be sequentially consistent. In this environment, threads can synchronize using ordinary memory operations. For example, a producer thread and a consumer thread can synchronize access to a circular data buffer as follows:

```
int rdptr = 0;
int wrptr = 0;
data_t buf[BUFSIZE];

Thread 1:
    while (work_to_do) {
        int next;

        buf[wrptr] = produce();
        next = (wrptr + 1) % BUFSIZE;
        while (rdptr == next)
            ;
        wrptr = next;
    }

Thread 2:
    while (work_to_do) {
        while (rdptr == wrptr)
            ;
        consume(buf[rdptr]);
        rdptr = (rdptr + 1) % BUFSIZE;
    }
```

In modern multi-processors, these conditions are relaxed to achieve greater performance. If one processor stores values in location A and then location B, then other processors loading data from location B and then location A may see the new value of B but the old value of A. The memory operations of such machines are said to be weakly ordered. On these machines, the circular buffer technique shown in the example will fail because the consumer may see the new value of *wrptr* but the old value of the data in the buffer. In such machines, synchronization can only be achieved through the use of special instructions that enforce an order on memory operations. Most high-level language compilers only generate ordinary memory operations to take advantage of the increased performance. They usually cannot determine when memory operation order is important and generate the special ordering instructions. Instead, they rely on the programmer to use synchronization primitives correctly to ensure that modifications to a location in memory are ordered with respect to modifications and/or access to the same location in other threads. Access to read-only data need not be synchronized. The resulting program is said to be data race-free.

Synchronization is still important even when accessing a single primitive variable (for example, an integer). On machines where the integer may not be aligned to the bus data width or be larger than the data width, a single memory load may require multiple memory cycles. This means that it may be possible for some parts of the integer to have an old value while other parts have a newer value. On some processor architectures this cannot happen, but portable programs cannot rely on this.

In summary, a portable multi-threaded program, or a multi-process program that shares writable memory between processes, has to use the synchronization primitives to synchronize data access. It cannot rely on modifications to memory being observed by other threads in the order written in the program or even on modification of a single variable being seen atomically.

Conforming applications may only use the functions listed to synchronize threads of control with respect to memory access. There are many other candidates for functions that might also be used. Examples are: signal sending and reception, or pipe writing and reading. In general, any function that allows one thread of control to wait for an action caused by another thread of control is a candidate. IEEE Std. 1003.1-200x does not require these additional functions to synchronize memory access since this would imply the following:

- All these functions would have to be recognized by advanced compilation systems so that memory operations and calls to these functions are not reordered by optimization.
- All these functions would potentially have to have memory synchronization instructions added, depending on the particular machine.
- The additional functions complicate the model of how memory is synchronized and make automatic data race detection techniques impractical.

Formal definitions of the memory model were rejected as unreadable by the vast majority of programmers. In addition, most of the formal work in the literature has concentrated on the memory as provided by the hardware as opposed to the application programmer through the compiler and runtime system. It was believed that a simple statement intuitive to most programmers would be most effective. IEEE Std. 1003.1-200x defines functions that can be used to synchronize access to memory, but it leaves open exactly how one relates those functions to the semantics of each function as specified elsewhere in IEEE Std. 1003.1-200x. IEEE Std. 1003.1-200x also does not make a formal specification of the partial ordering in time that the functions can impose, as that is implied in the description of the semantics of each function. It simply states that the programmer has to ensure that modifications do not occur ''simultaneously'' with other access to a memory location.

A.4.9 Path Name Resolution

It is necessary to differentiate between the definition of path name and the concept of path name resolution with respect to the handling of trailing slashes. By specifying the behavior here, it is not possible to provide an implementation that is conforming but extends all interfaces that handle path names to also handle strings that are not legal path names (because they have trailing slashes).

Path names that end with one or more trailing slash characters must refer to directory paths. Previous versions of IEEE Std. 1003.1-200x were not specific about the distinction between trailing slashes on files and directories, and both were permitted.

Two types of implementation have been prevalent; those that ignored trailing slash characters on all path names regardless, and those that only permitted them only on existing directories.

IEEE Std. 1003.1-200x requires that a path name with a trailing slash character be treated as if it had a trailing `"/."` everywhere.

Note that this change does not break any portable applications; since there were two different types of implementation, no application could have portably depended on either behavior. This change does however require some implementations to be altered to remain compliant. Substantial discussion over a three-year period has shown that the benefits to application developers outweighs the disadvantages for some vendors.

On a historical note, some early applications automatically appended a `'/'` to every path. Rather than fix the applications, the system implementation was modified to accept this behavior by ignoring any trailing slash.

Each directory has exactly one parent directory which is represented by the name **dot-dot** in the first directory. No other directory, regardless of linkages established by symbolic links, is considered the parent directory by IEEE Std. 1003.1-200x.

There are two general categories of interfaces involving path name resolution: those that follow the symbolic link, and those that do not. There are several exceptions to this rule; for example, *open*(*path*,O_CREAT | O_EXCL) will fail when *path* names a symbolic link. However, in all other situations, the *open*( ) function will follow the link.

What the file name **dot-dot** refers to relative to the root directory is implementation-defined. In Version 7 it refers to the root directory itself; this is the behavior mentioned in IEEE Std. 1003.1-200x. In some networked systems the construction **/../hostname/** is used to refer to the root directory of another host, and POSIX.1 permits this behavior.

Other networked systems use the construct **//hostname** for the same purpose; that is, a double initial slash is used. There is a potential problem with existing applications that create full path names by taking a trunk and a relative path name and making them into a single string separated by `'/'`, because they can accidentally create networked path names when the trunk is `'/'`. This practice is not prohibited because such applications can be made to conform by simply changing to use `"//"` as a separator instead of `'/'`:

- If the trunk is `'/'`, the full path name will begin with `"///"` (the initial `'/'` and the separator `"//"`). This is the same as `'/'`, which is what is desired. (This is the general case of making a relative path name into an absolute one by prefixing with `"///"` instead of `'/'`.)
- If the trunk is `"/A"`, the result is `"/A//..."`; since non-leading sequences of two or more slashes are treated as a single slash, this is equivalent to the desired `"/A/..."`.
- If the trunk is `"//A"`, the implementation-defined semantics will apply. (The multiple slash rule would apply.)

Application developers should avoid generating path names that start with `"//"`. Implementations are strongly encouraged to avoid using this special interpretation since a number of applications currently do not follow this practice and may inadvertently generate `"//..."`.

The term *root directory* is only defined in POSIX.1 relative to the process. In some implementations, there may be no absolute root directory. The initialization of the root directory of a process is implementation-defined.

A.4.10 Process ID Reuse

There is no additional rationale provided for this section.

A.4.11 Scheduling Policy

There is no additional rationale provided for this section.

A.4.12 Seconds Since the Epoch

There is no additional rationale provided for this section.

A.4.13 Semaphore

There is no additional rationale provided for this section.

A.4.14 Thread-Safety

Where the interface of a function required by IEEE Std. 1003.1-200x precludes thread-safety, an alternate form that shall be thread-safe is provided. The names of these thread-safe forms are the same as the non-thread-safe forms with the addition of the suffix ‘‘_r’’. The suffix ‘‘_r’’ is historical, where the `'r'` stood for ‘‘reentrant’’.

In some cases, thread-safety is provided by restricting the arguments to an existing function.

See also Section B.2.9.1 (on page 3460).

A.4.15 Utility

There is no additional rationale provided for this section.

A.4.16 Variable Assignment

There is no additional rationale provided for this section.

A.5 File Format Notation

The notation for spaces allows some flexibility for application output. Note that an empty character position in *format* represents one or more <blank> characters on the output (not *white space*, which can include <newline> characters). Therefore, another utility that reads that output as its input must be prepared to parse the data using *scanf*( ), *awk*, and so on. The `'Δ'` character is used when exactly one <space> character is output.

The treatment of integers and spaces is different from the *printf*( ) function in that they can be surrounded with <blank> characters. This was done so that, given a format such as:

```
"%d\n",<foo>
```

the implementation could use a *printf*( ) call such as:

```
printf("%6d\n", foo);
```

and still conform. This notation is thus somewhat like *scanf*( ) in addition to *printf*( ).

The *printf*( ) function was chosen as a model because most of the standard developers were familiar with it. One difference from the C function *printf*( ) is that the *l* and *h* conversion characters are not used. As expressed by the Shell and Utilities volume of IEEE Std. 1003.1-200x, there is no differentiation between decimal values for type **int**, type **long**, or type **short**. The specifications %*d* or %*i* should be interpreted as an arbitrary length sequence of digits. Also, no distinction is made between single precision and double precision numbers (**float** or **double** in C). These are simply referred to as floating point numbers.

Many of the output descriptions in the Shell and Utilities volume of IEEE Std. 1003.1-200x use the term *line*, such as:

```
"%s", <input line>
```

Since the definition of *line* includes the trailing <newline> character already, there is no need to include a `'\n'` in the format; a double <newline> character would otherwise result.

A.6 Character Set

A.6.1 Portable Character Set

The portable character set is listed in full so there is no dependency on the ISO / IEC 646: 1991 standard (or historically ASCII) encoded character set, although the set is identical to the characters defined in the International Reference version of the ISO / IEC 646: 1991 standard.

IEEE Std. 1003.1-200x poses no requirement that multiple character sets or codesets be supported, leaving this as a marketing differentiation for implementors. Although multiple charmap files are supported, it is the responsibility of the implementation to provide the file(s); if only one is provided, only that one will be accessible using the *localedef* **−f** option.

The statement about invariance in codesets for the portable character set is worded to avoid precluding implementations where multiple incompatible codesets are available (for instance, ASCII and EBCDIC). The standard utilities cannot be expected to produce predictable results if they access portable characters that vary on the same implementation.

Not all character sets need include the portable character set, but each locale must include it. For example, a Japanese-based locale might be supported by a mixture of character sets: JIS X 0201 Roman (a Japanese version of the ISO / IEC 646: 1991 standard), JIS X 0208, and JIS X 0201 Katakana. Not all of these character sets include the portable characters, but at least one does (JIS X 0201 Roman).

A.6.2 Character Encoding

Encoding mechanisms based on single shifts, such as the EUC encoding used in some Asian and other countries, can be supported via the current charmap mechanism. With single-shift encoding, each character is preceded by a shift code (SS2 or SS3). A complete EUC code, consisting of the portable character set (G0) and up to three additional character sets (G1, G2, G3), can be described using the current charmap mechanism; the encoding for each character in additional character sets G2 and G3 must then include their single-shift code. Other mechanisms to support locales based on encoding mechanisms such as locking shift are not addressed by this volume of IEEE Std. 1003.1-200x.

A.6.3 C Language Wide-Character Codes

There is no additional rationale for this section.

A.6.4 Character Set Description File

A.6.4.1 State-Dependent Character Encodings

A requirement was considered that would force utilities to eliminate any redundant locking shifts, but this was left as a quality of implementation issue.

This change satisfies the following requirement from the ISO POSIX-2: 1993 standard, Annex H.1:

The support of state-dependent (shift encoding) character sets should be addressed fully. See descriptions of these in the Base Definitions volume of IEEE Std. 1003.1-200x, Section 6.2, Character Encoding. If such character encodings are supported, it is expected that this will impact the Base Definitions volume of IEEE Std. 1003.1-200x, Section 6.2, Character Encoding, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 9, Regular Expressions , and the comm, cut, diff, grep, head, join, paste, and tail utilities.

The character set description file provides:

- The capability to describe character set attributes (such as collation order or character classes) independent of character set encoding, and using only the characters in the portable character set. This makes it possible to create generic *localedef* source files for all codesets that share the portable character set (such as the ISO 8859 family or IBM Extended ASCII).
- Standardized symbolic names for all characters in the portable character set, making it possible to refer to any such character regardless of encoding.

Implementations are free to choose their own symbolic names, as long as the names identified by this volume of IEEE Std. 1003.1-200x are also defined; this provides support for already existing ‘‘character names’’.

The names selected for the members of the portable character set follow the ISO/IEC 8859-1:1998 standard and the ISO/IEC 10646-1:1993 standard. However, several commonly used UNIX system names occur as synonyms in the list:

- The historical UNIX system names are used for control characters.
- The word ‘‘slash’’ is given in addition to ‘‘solidus’’.
- The word ‘‘backslash’’ is given in addition to ‘‘reverse-solidus’’.
- The word ‘‘hyphen’’ is given in addition to ‘‘hyphen-minus’’.
- The word ‘‘period’’ is given in addition to ‘‘full-stop’’.
- For digits, the word ‘‘digit’’ is eliminated.
- For letters, the words ‘‘Latin Capital Letter’’ and ‘‘Latin Small Letter’’ are eliminated.
- The words ‘‘left brace’’ and ‘‘right brace’’ are given in addition to ‘‘left-curly-bracket’’ and ‘‘right-curly-bracket’’.
- The names of the digits are preferred over the numbers to avoid possible confusion between `'0'` and `'O'`, and between `'1'` and `'l'` (one and the letter ell).

The names for the control characters in the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 6, Character Set were taken from the ISO/IEC 4873:1991 standard.

The charmap file was introduced to resolve problems with the portability of, especially, *localedef* sources. IEEE Std. 1003.1-200x assumes that the portable character set is constant across all locales, but does not prohibit implementations from supporting two incompatible codings, such as both ASCII and EBCDIC. Such dual-support implementations should have all charmaps and *localedef* sources encoded using one portable character set, in effect cross-compiling for the other environment. Naturally, charmaps (and *localedef* sources) are only portable without transformation between systems using the same encodings for the portable character set. They can, however, be transformed between two sets using only a subset of the actual characters (the portable character set). However, the particular coded character set used for an application or an implementation does not necessarily imply different characteristics or collation; on the contrary, these attributes should in many cases be identical, regardless of codeset. The charmap provides the capability to define a common locale definition for multiple codesets (the same *localedef* source can be used for codesets with different extended characters; the ability in the charmap to define empty names allows for characters missing in certain codesets).

The **<escape_char>** declaration was added at the request of the international community to ease the creation of portable charmap files on terminals not implementing the default backslash escape. The **<comment_char>** declaration was added at the request of the international community to eliminate the potential confusion between the number sign and the pound sign.

The octal number notation with no leading zero required was selected to match those of *awk* and *tr* and is consistent with that used by *localedef*. To avoid confusion between an octal constant and the back-references used in *localedef* source, the octal, hexadecimal, and decimal constants shall contain at least two digits. As single-digit constants are relatively rare, this should not impose any significant hardship. Provision is made for more digits to account for systems in which the byte size is larger than 8 bits. For example, a Unicode (ISO/IEC 10646-1: 1993 standard) system that has defined 16-bit bytes may require six octal, four hexadecimal, and five decimal digits.

The decimal notation is supported because some newer international standards define character values in decimal, rather than in the old column/row notation.

The charmap identifies the coded character sets supported by an implementation. At least one charmap shall be provided, but no implementation is required to provide more than one. Likewise, implementations can allow users to generate new charmaps (for instance, for a new version of the ISO 8859 family of coded character sets), but does not have to do so. If users are allowed to create new charmaps, the system documentation describes the rules that apply (for instance, ''only coded character sets that are supersets of the ISO/IEC 646: 1991 standard IRV, no multi-byte characters'').

This addition of the **WIDTH** specification satisfies the following requirement from the ISO POSIX-2: 1993 standard, Annex H.1:

(9) *The definition of column position relies on the implementation's knowledge of the integral width of the characters. The charmap or LC_CTYPE locale definitions should be enhanced to allow application specification of these widths.*

The character ''width'' information was first considered for inclusion under *LC_CTYPE* but was moved because it is more closely associated with the information in the *charmap* than information in the locale source (cultural conventions information). Concerns were raised that formalizing this type of information is moving the locale source definition from the codeset-independent entity that it was designed to be to a repository of codeset-specific information. A similar issue occurred with the **<code_set_name>**, **<mb_cur_max>**, and **<mb_cur_min>** information, which was resolved to reside in the *charmap* definition.

The width definition was added to the IEEE P1003.2b draft standard with the intent that the *wcswidth*( ) and/or *wcwidth*( ) functions (currently specified in the System Interfaces volume of IEEE Std. 1003.1-200x) be the mechanism to retrieve the character width information.

A.7 Locale

A.7.1 General

The description of locales is based on work performed in the UniForum Technical Committee Subcommittee on Internationalization. Wherever appropriate, keywords are taken from the ISO C standard or the X/Open Portability Guide.

The value used to specify a locale with environment variables is the name specified as the *name* operand to the *localedef* utility when the locale was created. This provides a verifiable method to create and invoke a locale.

The ‘‘object’’ definitions need not be portable, as long as ‘‘source’’ definitions are. Strictly speaking, source definitions are portable only between implementations using the same character set(s). Such source definitions, if they use symbolic names only, easily can be ported between systems using different codesets, as long as the characters in the portable character set (see the Base Definitions volume of IEEE Std. 1003.1-200x, Section 6.1, Portable Character Set) have common values between the codesets; this is frequently the case in historical implementations. Of source, this requires that the symbolic names used for characters outside the portable character set be identical between character sets. The definition of symbolic names for characters is outside the scope of IEEE Std. 1003.1-200x, but is certainly within the scope of other standards organizations.

Applications can select the desired locale by invoking the *setlocale*( ) function (or equivalent) with the appropriate value. If the function is invoked with an empty string, the value of the corresponding environment variable is used. If the environment variable is not set or is set to the empty string, the implementation sets the appropriate environment as defined in the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 8, Environment Variables.

A.7.2 POSIX Locale

The POSIX locale is equal to the C locale. To avoid being classified as a C-language function, the name has been changed to the POSIX locale; the environment variable value can be either `"POSIX"` or, for historical reasons, `"C"`.

The POSIX definitions mirror the historical UNIX system behavior.

The use of symbolic names for characters in the tables does not imply that the POSIX locale must be described using symbolic character names, but merely that it may be advantageous to do so.

A.7.3 Locale Definition

The decision to separate the file format from the *localedef* utility description was only partially editorial. Implementations may provide other interfaces than *localedef*. Requirements on ‘‘the utility’’, mostly concerning error messages, are described in this way because they are meant to affect the other interfaces implementations may provide as well as *localedef*.

The text about POSIX2_LOCALEDEF does not mean that internationalization is optional; only that the functionality of the *localedef* utility is. REs, for instance, must still be able to recognize, for example, character class expressions such as `"[[:alpha:]]"`. A possible analogy is with an applications development environment; while all conforming implementations must be capable of executing applications, not all need to have the development environment installed. The assumption is that the capability to modify the behavior of utilities (and applications) via locale settings must be supported. If the *localedef* utility is not present, then the only choice is to select an existing (presumably implementation-documented) locale. An implementation could, for example, choose to support only the POSIX locale, which would in effect limit the amount of changes from historical implementations quite drastically. The *localedef* utility is still required, but would always terminate with an exit code indicating that no locale could be created. Supported locales must be documented using the syntax defined in this chapter. (This ensures that users can accurately determine what capabilities are provided. If the implementation decides to provide additional capabilities to the ones in this chapter, that is already provided for.)

If the option is present (that is, locales can be created), then the *localedef* utility must be capable of creating locales based on the syntax and rules defined in this chapter. This does not mean that the implementation cannot also provide alternate means for creating locales.

The octal, decimal, and hexadecimal notations are the same employed by the charmap facility (see the Base Definitions volume of IEEE Std. 1003.1-200x, Section 6.4, Character Set Description File). To avoid confusion between an octal constant and a back-reference, the octal, hexadecimal, and decimal constants must contain at least two digits. As single-digit constants are relatively rare, this should not impose any significant hardship. Provision is made for more digits to account for systems in which the byte size is larger than 8 bits. For example, a Unicode (see the ISO / IEC 10646-1: 1993 standard) system that has defined 16-bit bytes may require six octal, four hexadecimal, and five decimal digits. As with the charmap file, multi-byte characters are described in the locale definition file using ‘‘big-endian’’ notation for reasons of portability. There is no requirement that the internal representation in the computer memory be in this same order.

One of the guidelines used for the development of this volume of IEEE Std. 1003.1-200x is that characters outside the invariant part of the ISO / IEC 646: 1991 standard should not be used in portable specifications. The backslash character is not in the invariant part; the number sign is, but with multiple representations: as a number sign, and as a pound sign. As far as general usage of these symbols, they are covered by the ‘‘grandfather clause’’, but for newly defined interfaces, the WG15 POSIX working group has requested that POSIX provide alternate representations. Consequently, while the default escape character remains the backslash and the default comment character is the number sign, implementations are required to recognize alternative representations, identified in the applicable source file via the **<escape_char>** and **<comment_char>** keywords.

A.7.3.1 LC_CTYPE

The *LC_CTYPE* category is primarily used to define the encoding-independent aspects of a character set, such as character classification. In addition, certain encoding-dependent characteristics are also defined for an application via the *LC_CTYPE* category. IEEE Std. 1003.1-200x does not mandate that the encoding used in the locale is the same as the one used by the application because an implementation may decide that it is advantageous to define locales in a system-wide encoding rather than having multiple, logically identical locales in different encodings, and to convert from the application encoding to the system-wide encoding on usage. Other implementations could require encoding-dependent locales.

In either case, the *LC_CTYPE* attributes that are directly dependent on the encoding, such as **<mb_cur_max>** and the display width of characters, are not user-specifiable in a locale source and are consequently not defined as keywords.

Implementations may define additional keywords or extend the *LC_CTYPE* mechanism to allow application-defined keywords.

The text ‘‘The ellipsis specification shall only be valid within a single encoded character set’’ is present because it is possible to have a locale supported by multiple character encodings, as explained in the rationale for the Base Definitions volume of IEEE Std. 1003.1-200x, Section 6.1, Portable Character Set. An example given there is of a possible Japanese-based locale supported by a mixture of the character sets JIS X 0201 Roman, JIS X 0208, and JIS X 0201 Katakana. Attempting to express a range of characters across these sets is not logical and the implementation is free to reject such attempts.

As the *LC_CTYPE* character classes are based on the ISO C standard character class definition, the category does not support multi-character elements. For instance, the German character <sharp-s> is traditionally classified as a lowercase letter. There is no corresponding uppercase letter; in proper capitalization of German text, the <sharp-s> will be replaced by "SS"; that is, by two characters. This kind of conversion is outside the scope of the **toupper** and **tolower** keywords.

Where IEEE Std. 1003.1-200x specifies that only certain characters can be specified, as for the keywords **digit** and **xdigit**, the specified characters shall be from the portable character set, as shown. As an example, only the Arabic digits 0 through 9 are acceptable as digits.

The character classes **digit**, **xdigit**, **lower**, **upper**, and **space** have a set of automatically included characters. These only need to be specified if the character values (that is, encoding) differs from the implementation default values. It is not possible to define a locale without these automatically included characters unless some implementation extension is used to prevent their inclusion. Such a definition would not be a proper superset of the C locale, and thus, it might not be possible for the standard utilities to be implemented as programs conforming to the ISO C standard.

The definition of character class **digit** requires that only ten characters—the ones defining digits—can be specified; alternate digits (for example, Hindi or Kanji) cannot be specified here. However, the encoding may vary if an implementation supports more than one encoding.

The definition of character class **xdigit** requires that the characters included in character class **digit** are included here also and allows for different symbols for the hexadecimal digits 10 through 15.

The inclusion of the **charclass** keyword satisfies the following requirement from the ISO POSIX-2: 1993 standard, Annex H.1:

> (3) *The LC_CTYPE (2.5.2.1) locale definition should be enhanced to allow user-specified additional character classes, similar in concept to the ISO C standard Multibyte Support Extension (MSE) is_wctype( ) function.*

This keyword was previously included in The Open Group specifications and is now mandated in the Shell and Utilities volume of IEEE Std. 1003.1-200x.

The symbolic constant {CHARCLASS_NAME_MAX} was also adopted from The Open Group specifications. Application portability is enhanced by the use of symbolic constants.

A.7.3.2 LC_COLLATE

The rules governing collation depend to some extent on the use. At least five different levels of increasingly complex collation rules can be distinguished:

1. *Byte/machine code order*: This is the historical collation order in the UNIX system and many proprietary operating systems. Collation is here performed character by character, without any regard to context. The primary virtue is that it usually is quite fast and also completely deterministic; it works well when the native machine collation sequence matches the user expectations.

2. *Character order*: On this level, collation is also performed character by character, without regard to context. The order between characters is, however, not determined by the code values, but on the expectations by the user of the ''correct'' order between characters. In

addition, such a (simple) collation order can specify that certain characters collate equally (for example, uppercase and lowercase letters).

3. *String ordering*: On this level, entire strings are compared based on relatively straightforward rules. Several ''passes'' may be required to determine the order between two strings. Characters may be ignored in some passes, but not in others; the strings may be compared in different directions; and simple string substitutions may be performed before strings are compared. This level is best described as ''dictionary'' ordering; it is based on the spelling, not the pronunciation, or meaning, of the words.

4. *Text search ordering*: This is a further refinement of the previous level, best described as ''telephone book ordering''; some common homonyms (words spelled differently but with the same pronunciation) are collated together; numbers are collated as if they were spelled out, and so on.

5. *Semantic-level ordering*: Words and strings are collated based on their meaning; entire words (such as ''the'') are eliminated; the ordering is not deterministic. This usually requires special software and is highly dependent on the intended use.

While the historical collation order formally is at level 1, for the English language it corresponds roughly to elements at level 2. The user expects to see the output from the *ls* utility sorted very much as it would be in a dictionary. While telephone book ordering would be an optimal goal for standard collation, this was ruled out as the order would be language-dependent. Furthermore, a requirement was that the order must be determined solely from the text string and the collation rules; no external information (for example, ''pronunciation dictionaries'') could be required.

As a result, the goal for the collation support is at level 3. This also matches the requirements for the Canadian collation order, as well as other, known collation requirements for alphabetic scripts. It specifically rules out collation based on pronunciation rules or based on semantic analysis of the text.

The syntax for the *LC_COLLATE* category source meets the requirements for level 3 and has been verified to produce the correct result with examples based on French, Canadian, and Danish collation order. Because it supports multi-character collating elements, it is also capable of supporting collation in codesets where a character is expressed using non-spacing characters followed by the base character (such as the ISO/IEC 6937:1994 standard).

The directives that can be specified in an operand to the **order_start** keyword are based on the requirements specified in several proposed standards and in customary use. The following is a rephrasing of rules defined for ''lexical ordering in English and French'' by the Canadian Standards Association (the text in square brackets is rephrased):

- Once special characters [punctuation] have been removed from original strings, the ordering is determined by scanning forwards (left to right) [disregarding case and diacriticals].

- In case of equivalence, special characters are once again removed from original strings and the ordering is determined by scanning backwards (starting from the rightmost character of the string and back), character by character [disregarding case but considering diacriticals].

- In case of repeated equivalence, special characters are removed again from original strings and the ordering is determined by scanning forwards, character by character [considering both case and diacriticals].

- If there is still an ordering equivalence after the first three rules have been applied, then only special characters and the position they occupy in the string are considered to determine ordering. The string that has a special character in the lowest position comes first. If two strings have a special character in the same position, the character [with the lowest collation value] comes first. In case of equality, the other special characters are considered until there is a difference or until all special characters have been exhausted.

It is estimated that this part of IEEE Std. 1003.1-200x covers the requirements for all European languages, and no particular problems are anticipated with Slavic or Middle East character sets.

The Far East (particularly Japanese/Chinese) collations are often based on contextual information and pronunciation rules (the same ideogram can have different meanings and different pronunciations). Such collation, in general, falls outside the desired goal of IEEE Std. 1003.1-200x. There are, however, several other collation rules (stroke/radical or ''most common pronunciation'') that can be supported with the mechanism described here.

The character (and collating element) order is defined by the order in which characters and elements are specified between the **order_start** and **order_end** keywords. This character order is used in range expressions in REs (see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 9, Regular Expressions). Weights assigned to the characters and elements define the collation sequence; in the absence of weights, the character order is also the collation sequence.

A.7.3.3 LC_MONETARY

The currency symbol does not appear in *LC_MONETARY* because it is not defined in the C locale of the ISO C standard.

The ISO C standard limits the size of decimal points and thousands delimiters to single-byte values. In locales based on multi-byte coded character sets, this cannot be enforced; IEEE Std. 1003.1-200x does not prohibit such characters, but makes the behavior unspecified (in the text ''In contexts where other standards ...'').

The grouping specification is based on, but not identical to, the ISO C standard. The −1 signals that no further grouping shall be performed; the equivalent of {CHAR_MAX} in the ISO C standard.

The text ''the value is not available in the locale'' is taken from the ISO C standard and is used instead of the ''unspecified'' text in early proposals. There is no implication that omitting these keywords or assigning them values of " " or −1 produces unspecified results; such omissions or assignments eliminate the effects described for the keyword or produce zero-length strings, as appropriate.

The locale definition is an extension of the ISO C standard *localeconv*( ) specification. In particular, rules on how **currency_symbol** is treated are extended to also cover **int_curr_symbol**, and **p_set_by_space** and **n_sep_by_space** have been augmented with the value 2, which places a <space> between the sign and the symbol (if they are adjacent; otherwise, it should be treated as a 0).

A.7.3.4 LC_NUMERIC

See the rationale for *LC_MONETARY* for a description of the behavior of grouping.

A.7.3.5 LC_TIME

Although certain of the field descriptors in the POSIX locale (such as the name of the month) are shown with initial capital letters, this need not be the case in other locales. Programs using these fields may need to adjust the capitalization if the output is going to be used at the beginning of a sentence.

The *LC_TIME* descriptions of **abday**, **day**, **mon**, and **abmon** imply a Gregorian style calendar (7-day weeks, 12-month years, leap years, and so on). Formatting time strings for other types of calendars is outside the scope of IEEE Std. 1003.1-200x.

While the ISO 8601:1988 standard numbers the weekdays starting with Monday, historical practice is to use the Sunday as the first day. Rather than change the order and introduce potential confusion, the days must be specified beginning with Sunday; previous references to ''first day'' have been removed. Note also that the Shell and Utilities volume of IEEE Std. 1003.1-200x *date* utility supports numbering compliant with the ISO 8601:1988 standard.

As specified under *date* in the Shell and Utilities volume of IEEE Std. 1003.1-200x and *strftime*( ) in the System Interfaces volume of IEEE Std. 1003.1-200x, the field descriptors corresponding to the optional keywords consist of a modifier followed by a traditional field descriptor (for instance %*Ex*). If the optional keywords are not supported by the implementation or are unspecified for the current locale, these field descriptors are treated as the traditional field descriptor. For example, assume the following keywords:

```
alt_digits   "0th";"1st";"2nd";"3rd";"4th";"5th";\
             "6th";"7th";"8th";"9th";"10th"

d_fmt        "The %Od day of %B in %Y"
```

On 7/4/1776, the %*x* field descriptor would result in `"The 4th day of July in 1776"`, while on 7/14/1789 would result in `"The 14 day of July in 1789"`. It can be noted that the above example is for illustrative purposes only; the %*O* modifier is primarily intended to provide for Kanji or Hindi digits in *date* formats.

A.7.3.6 LC_MESSAGES

A.7.4 Locale Definition Grammar

There is no additional rationale for this section.

A.7.4.1 Locale Lexical Conventions

There is no additional rationale for this section.

A.7.4.2 Locale Grammar

There is no additional rationale for this section.

A.7.5 Locale Definition Example

There is no additional rationale for this section.

A.8 Environment Variables

A.8.1 Environment Variable Definition

The variable *environ* is not intended to be declared in any header, but rather to be declared by the user for accessing the array of strings that is the environment. This is the traditional usage of the symbol. Putting it into a header could break some programs that use the symbol for their own purposes.

The decision to restrict conforming systems to the use of digits, uppercase letters, and underscores for environment variable names allows applications to use lowercase letters in their environment variable names without conflicting with any conforming system.

A.8.2 Internationalization Variables

The text about locale implies that any utilities written in standard C and conforming to IEEE Std. 1003.1-200x must issue the following call:

```
setlocale(LC_ALL, "")
```

If this were omitted, the ISO C standard specifies that the C locale would be used.

If any of the environment variables are invalid, it makes sense to default to an implementation-defined, consistent locale environment. It is more confusing for a user to have partial settings occur in case of a mistake. All utilities would then behave in one language/cultural environment. Furthermore, it provides a way of forcing the whole environment to be the implementation-defined default. Disastrous results could occur if a pipeline of utilities partially uses the environment variables in different ways. In this case, it would be appropriate for utilities that use *LANG* and related variables to exit with an error if any of the variables are invalid. For example, users typing individual commands at a terminal might want *date* to work if *LC_MONETARY* is invalid as long as *LC_TIME* is valid. Since these are conflicting reasonable alternatives, IEEE Std. 1003.1-200x leaves the results unspecified if the locale environment variables would not produce a complete locale matching the specification of the user.

The locale settings of individual categories cannot be truly independent and still guarantee correct results. For example, when collating two strings, characters must first be extracted from each string (governed by *LC_CTYPE*) before being mapped to collating elements (governed by *LC_COLLATE*) for comparison. That is, if *LC_CTYPE* is causing parsing according to the rules of a large, multi-byte code set (potentially returning 20 000 or more distinct character codeset values), but *LC_COLLATE* is set to handle only an 8-bit codeset with 256 distinct characters, meaningful results are obviously impossible.

The *LC_MESSAGES* variable affects the language of messages generated by the standard utilities.

The description of the environment variable names starting with the characters ‘‘LC_’’ acknowledges the fact that the interfaces presented may be extended as new international functionality is required. In the ISO C standard, names preceded by ‘‘LC_’’ are reserved in the name space for future categories.

To avoid name clashes, new categories and environment variables are divided into two classifications: *implementation-independent* and *implementation-defined*.

Implementation-independent names will have the following format:

LC_*NAME*

where *NAME* is the name of the new category and environment variable. Capital letters must be used for implementation-independent names.

Implementation-defined names must be in lowercase letters, as below:

`LC_`*`name`*

A.8.3 Other Environment Variables

The quoted form of the timezone variable allows timezone names of the form UTC+1 (or any name that contains the character plus (`'+'`), the character minus (`'−'`), or digits), which may be appropriate for countries that do not have an official timezone name. It would be coded as <UTC+1>+1<UTC+2>, which would cause *std* to have a value of UTC+1 and *dst* a value of UTC+2, each with a length of 6 characters. This does not appear to conflict with any existing usage. The characters `'<'` and `'>'` were chosen for quoting because they are easier to parse visually than a quoting character that does not provide some sense of bracketing (and in a string like this, such bracketing is helpful). They were also chosen because they do not need special treatment when assigning to the *TZ* variable. Users are often confused by embedding quotes in a string. Because `'<'` and `'>'` are meaningful to the shell, the whole string would have to be quoted, but that is easily explained. (Parentheses would have presented the same problems.) Although the `'>'` symbol could have been permitted in the string by either escaping it or doubling it, it seemed of little value to require that. This could be provided as an extension if there was a need. Timezone names of this new form lead to a requirement that the value of {_POSIX_TZNAME_MAX} change from 3 to 6.

COLUMNS, LINES

The default value for the number of column positions, *COLUMNS*, and screen height, *LINES*, are unspecified because historical implementations use different methods to determine values corresponding to the size of the screen in which the utility is run. This size is typically known to the implementation through the value of *TERM*, or by more elaborate methods such as extensions to the *stty* utility or knowledge of how the user is dynamically resizing windows on a bit-mapped display terminal. Users should not need to set these variables in the environment unless there is a specific reason to override the default behavior of the implementation, such as to display data in an area arbitrarily smaller than the terminal or window. Values for these variables that are not decimal integers greater than zero are implicitly undefined values; it is unnecessary to enumerate all of the possible values outside of the acceptable set.

PATH

Many historical implementations of the Bourne shell do not interpret a trailing colon to represent the current working directory and are thus non-conforming. The C Shell and the KornShell conform to IEEE Std. 1003.1-200x on this point. The usual name of dot may also be used to refer to the current working directory.

Many implementations historically have used a default value of **/bin** and **/usr/bin** for the *PATH* variable. IEEE Std. 1003.1-200x does not mandate this default path be identical to that retrieved from *getconf* _CS_PATH because it is likely that the standardized utilities may be provided in another directory separate from the directories used by some historical applications.

LOGNAME

In most implementations, the value of such a variable is easily forged, so security-critical applications should rely on other means of determining user identity. *LOGNAME* is required to be constructed from the portable file name character set for reasons of interchange. No diagnostic condition is specified for violating this rule, and no requirement for enforcement exists. The intent of the requirement is that if extended characters are used, the ''guarantee'' of portability implied by a standard is void.

SHELL

The *SHELL* variable names the preferred shell of the user; it is a guide to applications. There is no direct requirement that that shell conform to IEEE Std. 1003.1-200x; that decision should rest with the user. It is the intention of the standard developers that alternative shells be permitted, if the user chooses to develop or acquire one. An operating system that builds its shell into the ''kernel'' in such a manner that alternative shells would be impossible does not conform to the spirit of IEEE Std. 1003.1-200x.

CHANGE HISTORY

Issue 6

Changed format of *TZ* field to allow for the quoted form as defined in previous versions of the ISO POSIX-1 standard.

A.9 Regular Expressions

Rather than repeating the description of REs for each utility supporting REs, the standard developers preferred a common, comprehensive description of regular expressions in one place. The most common behavior is described here, and exceptions or extensions to this are documented for the respective utilities, as appropriate.

The BRE corresponds to the *ed* or historical *grep* type, and the ERE corresponds to the historical *egrep* type (now *grep* –**E**).

The text is based on the *ed* description and substantially modified, primarily to aid developers and others in the understanding of the capabilities and limitations of REs. Much of this was influenced by internationalization requirements.

It should be noted that the definitions in this section do not cover the *tr* utility; the *tr* syntax does not employ REs.

The specification of REs is particularly important to internationalization because pattern matching operations are very basic operations in business and other operations. The syntax and rules of REs are intended to be as intuitive as possible to make them easy to understand and use. The historical rules and behavior do not provide that capability to non-English language users, and do not provide the necessary support for commonly used characters and language constructs. It was necessary to provide extensions to the historical RE syntax and rules to accommodate other languages.

As they are limited to bracket expressions, the rationale for these modifications is in the Base Definitions volume of IEEE Std. 1003.1-200x, Section 9.3.5, RE Bracket Expression.

A.9.1 Regular Expression Definitions

It is possible to determine what strings correspond to subexpressions by recursively applying the leftmost longest rule to each subexpression, but only with the proviso that the overall match is leftmost longest. For example, matching `"\(ac*\)c*d[ac]*\1"` against *acdacaaa* matches *acdacaaa* (with \1=*a*); simply matching the longest match for `"\(ac*\)"` would yield \1=*ac*, but the overall match would be smaller (*acdac*). Conceptually, the implementation must examine every possible match and among those that yield the leftmost longest total matches, pick the one that does the longest match for the leftmost subexpression, and so on. Note that this means that matching by subexpressions is context-dependent: a subexpression within a larger RE may match a different string from the one it would match as an independent RE, and two instances of the same subexpression within the same larger RE may match different lengths even in similar sequences of characters. For example, in the ERE `"(a.*b)(a.*b)"`, the two identical subexpressions would match four and six characters, respectively, of *accbaccccb*.

The definition of *single character* has been expanded to include also collating elements consisting of two or more characters; this expansion is applicable only when a bracket expression is included in the BRE or ERE. An example of such a collating element may be the Dutch *ij*, which collates as a `'y'`. In some encodings, a ligature ‘‘i with j’’ exists as a character and would represent a single-character collating element. In another encoding, no such ligature exists, and the two-character sequence *ij* is defined as a multi-character collating element. Outside brackets, the *ij* is treated as a two-character RE and matches the same characters in a string. Historically, a bracket expression only matched a single character. If, however, the bracket expression defines, for example, a range that includes *ij*, then this particular bracket expression also matches a sequence of the two characters `'i'` and `'j'` in the string.

A.9.2 Regular Expression General Requirements

The definition of which sequence is matched when several are possible is based on the leftmost-longest rule historically used by deterministic recognizers. This rule is easier to define and describe, and arguably more useful, than the first-match rule historically used by non-deterministic recognizers. It is thought that dependencies on the choice of rule are rare; carefully contrived examples are needed to demonstrate the difference.

A formal expression of the leftmost-longest rule is:

> The search is performed as if all possible suffixes of the string were tested for a prefix matching the pattern; the longest suffix containing a matching prefix is chosen, and the longest possible matching prefix of the chosen suffix is identified as the matching sequence.

Historically, most RE implementations only match lines, not strings. However, that is more an effect of the usage than of an inherent feature of REs themselves. Consequently, IEEE Std. 1003.1-200x does not regard <newline>s as special; they are ordinary characters, and both a period and a non-matching list can match them. Those utilities (like *grep*) that do not allow <newline>s to match are responsible for eliminating any <newline> from strings before matching against the RE. The *regcomp*( ) function, however, can provide support for such processing without violating the rules of this section.

The definition of case-insensitive processing is intended to allow matching of multi-character collating elements as well as characters. For instance, as each character in the string is matched using both its cases, the RE `"[[.Ch.]]"`, when matched against `"char"`, is in reality matched against `"ch"`, `"Ch"`, `"cH"`, and `"CH"`.

Some implementations of *egrep* have had very limited flexibility in handling complex EREs. IEEE Std. 1003.1-200x does not attempt to define the complexity of a BRE or ERE, but does place a lower limit on it—any RE must be handled, as long as it can be expressed in 256 bytes or less. (Of course, this does not place an upper limit on the implementation.) There are historical programs using a non-deterministic-recognizer implementation that should have no difficulty with this limit. It is possible that a good approach would be to attempt to use the faster, but more limited, deterministic recognizer for simple expressions and to fall back on the non-deterministic recognizer for those expressions requiring it. Non-deterministic implementations must be careful to observe the rules on which match is chosen; the longest match, not the first match, starting at a given character is used.

The term *invalid* highlights a difference between this section and some others: IEEE Std. 1003.1-200x frequently avoids mandating of errors for syntax violations because they can be used by implementors to trigger extensions. However, the authors of the internationalization features of REs wanted to mandate errors for certain conditions to identify usage problems or non-portable constructs. These are identified within this rationale as appropriate. The remaining syntax violations have been left implicitly or explicitly undefined. For example, the BRE construct `"\{1,2,3\}"` does not comply with the grammar. A conforming application cannot rely on it producing an error nor matching the literal characters `"\{1,2,3\}"`. The term ‘‘undefined’’ was used in favor of ‘‘unspecified’’ because many of the situations are considered errors on some implementations, and the standard developers considered that consistency throughout the section was preferable to mixing undefined and unspecified.

A.9.3 Basic Regular Expressions

There is no additional rationale for this section.

A.9.3.1 BREs Matching a Single Character or Collating Element

There is no additional rationale for this section.

A.9.3.2 BRE Ordinary Characters

There is no additional rationale for this section.

A.9.3.3 BRE Special Characters

There is no additional rationale for this section.

A.9.3.4 Periods in BREs

There is no additional rationale for this section.

A.9.3.5 RE Bracket Expression

Range expressions are, historically, an integral part of REs. However, the requirements of ‘‘natural language behavior’’ and portability do conflict: ranges must be treated according to the current collating sequence and include such characters that fall within the range based on that collating sequence, regardless of character values. This means, however, that the interpretation will differ depending on collating sequence. If, for instance, one collating sequence defines `'a` as a variant of `'a'`, while another defines it as a letter following `'z'`, then the expression `"[a–z]"` is valid in the first language and invalid in the second. This kind of ambiguity should be avoided in portable applications, and therefore the standard developers elected to state that ranges must not be used in strictly conforming applications; however, implementations must support them.

Some historical implementations allow range expressions where the ending range point of one range is also the starting point of the next (for instance, `"[a–m–o]"`). This behavior should not be permitted, but to avoid breaking historical implementations, it is now *undefined* whether it is a valid expression and how it should be interpreted.

Current practice in *awk* and *lex* is to accept escape sequences in bracket expressions as per the Base Definitions volume of IEEE Std. 1003.1-200x, Table 5-1, Escape Sequences and Associated Actions, while the normal ERE behavior is to regard such a sequence as consisting of two characters. Allowing the *awk*/*lex* behavior in EREs would change the normal behavior in an unacceptable way; it is expected that *awk* and *lex* will decode escape sequences in EREs before passing them to *regcomp*( ) or comparable routines. Each utility describes the escape sequences it accepts as an exception to the rules in this section; the list is not the same, for historical reasons.

As noted previously, the new syntax and rules have been added to accommodate other languages than English. The remainder of this section describes the rationale for these modifications.

A.9.3.6 BREs Matching Multiple Characters

The limit of nine back-references to subexpressions in the RE is based on the use of a single-digit identifier; increasing this to multiple digits would break historical applications. This does not imply that only nine subexpressions are allowed in REs. The following is a valid BRE with ten subexpressions:

```
\(\(\(ab\)*c\)*d\)\(ef\)*\(gh\)\{2\}\(ij\)*\(kl\)*\(mn\)*\(op\)*\(qr\)*
```

The standard developers regarded the common historical behavior, which supported `"\n*"`, but not `"\n\{min,max\}"`, `"\(...\)*"`, or `"\(...\)\{min,max\}"`, as a non-intentional result of a specific implementation, and they supported both duplication and interval expressions following subexpressions and back-references.

The changes to the processing of the back-reference expression remove an unspecified or ambiguous behavior in the Shell and Utilities volume of IEEE Std. 1003.1-200x, aligning it with the requirements specified for the *regcomp*( ) expression, and is the result of PASC Interpretation 1003.2-92 #43 submitted for the ISO POSIX-2: 1993 standard.

A.9.3.7 BRE Precedence

There is no additional rationale for this section.

A.9.3.8 BRE Expression Anchoring

Often, the dollar sign is viewed as matching the ending <newline> in text files. This is not strictly true; the <newline> is typically eliminated from the strings to be matched, and the dollar sign matches the terminating null character.

The ability of `'^'`, `'$'`, and `'*'` to be non-special in certain circumstances may be confusing to some programmers, but this situation was changed only in a minor way from historical practice to avoid breaking many historical scripts. Some consideration was given to making the use of the anchoring characters undefined if not escaped and not at the beginning or end of strings. This would cause a number of historical BREs, such as `"2^10"`, `"$HOME"`, and `"$1.35"`, that relied on the characters being treated literally, to become invalid.

However, one relatively uncommon case was changed to allow an extension used on some implementations. Historically, the BREs `"^foo"` and `"\(^foo\)"` did not match the same string, despite the general rule that subexpressions and entire BREs match the same strings. To increase consensus, IEEE Std. 1003.1-200x has allowed an extension on some systems to treat these two cases in the same way by declaring that anchoring *may* occur at the beginning or end of a subexpression. Therefore, portable BREs that require a literal circumflex at the beginning or a dollar sign at the end of a subexpression must escape them. Note that a BRE such as `"a\(^bc\)"` will either match `"a^bc"` or nothing on different systems under the rules.

ERE anchoring has been different from BRE anchoring in all historical systems. An unescaped anchor character has never matched its literal counterpart outside a bracket expression. Some systems treated `"foo$bar"` as a valid expression that never matched anything; others treated it as invalid. IEEE Std. 1003.1-200x mandates the former, valid unmatched behavior.

Some systems have extended the BRE syntax to add alternation. For example, the subexpression `"\(foo$\|bar\)"` would match either `"foo"` at the end of the string or `"bar"` anywhere. The extension is triggered by the use of the undefined `"\|"` sequence. Because the BRE is undefined for portable scripts, the extending system is free to make other assumptions, such that the `'$'` represents the end-of-line anchor in the middle of a subexpression. If it were not for the extension, the `'$'` would match a literal dollar sign under the rules.

A.9.4 Extended Regular Expressions

As with BREs, the standard developers decided to make the interpretation of escaped ordinary characters undefined.

The right parenthesis is not listed as an ERE special character because it is only special in the context of a preceding left parenthesis. If found without a preceding left parenthesis, the right parenthesis has no special meaning.

The *interval expression*, `"{m,n}"`, has been added to EREs. Historically, the interval expression has only been supported in some ERE implementations. The standard developers estimated that the addition of interval expressions to EREs would not decrease consensus and would also make BREs more of a subset of EREs than in many historical implementations.

It was suggested that, in addition to interval expressions, back-references (`'\n'`) should also be added to EREs. This was rejected by the standard developers as likely to decrease consensus.

In historical implementations, multiple duplication symbols are usually interpreted from left to right and treated as additive. As an example, `"a+*b"` matches zero or more instances of `'a'` followed by a `'b'`. In IEEE Std. 1003.1-200x, multiple duplication symbols are undefined; that is, they cannot be relied upon for portable applications. One reason for this is to provide some scope for future enhancements.

The precedence of operations differs between EREs and those in *lex*; in *lex*, for historical reasons, interval expressions have a lower precedence than concatenation.

A.9.4.1 EREs Matching a Single Character or Collating Element

There is no additional rationale for this section.

A.9.4.2 ERE Ordinary Characters

There is no additional rationale for this section.

A.9.4.3 ERE Special Characters

There is no additional rationale for this section.

A.9.4.4 Periods in EREs

There is no additional rationale for this section.

A.9.4.5 ERE Bracket Expression

There is no additional rationale for this section.

A.9.4.6 EREs Matching Multiple Characters

There is no additional rationale for this section.

A.9.4.7 ERE Alternation

There is no additional rationale for this section.

A.9.4.8 ERE Precedence

There is no additional rationale for this section.

A.9.4.9 ERE Expression Anchoring

There is no additional rationale for this section.

A.9.5 Regular Expression Grammar

The grammars are intended to represent the range of acceptable syntaxes available to portable applications. There are instances in the text where undefined constructs are described; as explained previously, these allow implementation extensions. There is no intended requirement that an implementation extension must somehow fit into the grammars shown here.

The BRE grammar does not permit L_ANCHOR or R_ANCHOR inside `"\("` and `"\)"` (which implies that `'^'` and `'$'` are ordinary characters). This reflects the semantic limits on the application, as noted in the Base Definitions volume of IEEE Std. 1003.1-200x, Section 9.3.8, BRE Expression Anchoring. Implementations are permitted to extend the language to interpret `'^'` and `'$'` as anchors in these locations, and as such, portable applications cannot use unescaped `'^'` and `'$'` in positions inside `"\("` and `"\)"` that might be interpreted as anchors.

The ERE grammar does not permit several constructs that the Base Definitions volume of IEEE Std. 1003.1-200x, Section 9.4.2, ERE Ordinary Characters and the Base Definitions volume of IEEE Std. 1003.1-200x, Section 9.4.3, ERE Special Characters specify as having undefined results:

- ORD_CHAR preceded by `'\'`
- *ERE_dupl_symbol*(s) appearing first in an ERE, or immediately following `'|'`, `'^'`, or `'('`
- `'{'` not part of a valid *ERE_dupl_symbol*
- `'|'` appearing first or last in an ERE, or immediately following `'|'` or `'('`, or immediately preceding `')'`

Implementations are permitted to extend the language to allow these. Portable applications cannot use such constructs.

A.9.5.1 BRE/ERE Grammar Lexical Conventions

There is no additional rationale for this section.

A.9.5.2 RE and Bracket Expression Grammar

The removal of the *Back_open_paren Back_close_paren* option from the *nondupl_RE* specification is the result of PASC Interpretation 1003.2-92 #43 submitted for the ISO POSIX-2: 1993 standard. Although the grammar required support for null subexpressions, this section does not describe the meaning of, and historical practice did not support, this construct.

A.9.5.3 ERE Grammar

There is no additional rationale for this section.

A.10 Directory Structure and Devices

A.10.1 Directory Structure and Files

A description of the historical **/usr/tmp** was omitted, removing any concept of differences in emphasis between the **/** and **/usr** directories. The descriptions of **/bin**, **/usr/bin**, **/lib**, and **/usr/lib** were omitted because they are not useful for applications. In an early draft, a distinction was made between system and application directory usage, but this was not found to be useful.

The directories **/** and **/dev** are included because the notion of a hierarchical directory structure is key to other information presented elsewhere in IEEE Std. 1003.1-200x. In early drafts, it was argued that special devices and temporary files could conceivably be handled without a directory structure on some implementations. For example, the system could treat the characters `"/tmp"` as a special token that would store files using some non-POSIX file system structure. This notion was rejected by the standard developers, who required that all the files in this section be implemented via POSIX file systems.

The **/tmp** directory is retained in IEEE Std. 1003.1-200x to accommodate historical applications that assume its availability. Implementations are encouraged to provide suitable directory names in the environment variable *TMPDIR* and applications are encouraged to use the contents of *TMPDIR* for creating temporary files.

The standard files **/dev/null** and **/dev/tty** are required to be both readable and writable to allow applications to have the intended historical access to these files.

The standard file **/dev/console** has been added for alignment with the Single UNIX Specification.

A.10.2 Output Devices and Terminal Types

There is no additional rationale for this section.

A.11 General Terminal Interface

If the implementation does not support this interface on any device types, it should behave as if it were being used on a device that is not a terminal device (in most cases *errno* will be set to [ENOTTY] on return from functions defined by this interface). This is based on the fact that many applications are written to run both interactively and in some non-interactive mode, and they adapt themselves at runtime. Requiring that they all be modified to test an environment variable to determine whether they should try to adapt is unnecessary. On a system that provides no general terminal interface, providing all the entry points as stubs that return [ENOTTY] (or an equivalent, as appropriate) has the same effect and requires no changes to the application.

Although the needs of both interface implementors and application developers were addressed throughout IEEE Std. 1003.1-200x, this section pays more attention to the needs of the latter. This is because, while many aspects of the programming interface can be hidden from the user by the application developer, the terminal interface is usually a large part of the user interface. Although to some extent the application developer can build missing features or work around inappropriate ones, the difficulties of doing that are greater in the terminal interface than elsewhere. For example, efficiency prohibits the average program from interpreting every character passing through it in order to simulate character erase, line kill, and so on. These functions should usually be done by the operating system, possibly at the interrupt level.

The *tc**( ) functions were introduced as a way of avoiding the problems inherent in the traditional *ioctl*( ) function and in variants of it that were proposed. For example, *tcsetattr*( ) is specified in place of the use of the TCSETA *ioctl*( ) command function. This allows specification of all the arguments in a manner consistent with the ISO C standard unlike the varying third argument of *ioctl*( ), which is sometimes a pointer (to any of many different types) and sometimes an **int**.

The advantages of this new method include:

- It allows strict type checking.
- The direction of transfer of control data is explicit.
- Portable capabilities are clearly identified.
- The need for a general interface routine is avoided.
- Size of the argument is well-defined (there is only one type).

The disadvantages include:

- No historical implementation uses the new method.
- There are many small routines instead of one general-purpose one.
- The historical parallel with *fcntl*( ) is broken.

The issue of modem control was excluded from IEEE Std. 1003.1-200x on the grounds that:

- It was concerned with setting and control of hardware timers.
- The appropriate timers and settings vary widely internationally.
- Feedback from European computer manufacturers indicated that this facility was not consistent with European needs and that specification of such a facility was not a requirement for portability.

A.11.1 Interface Characteristics

A.11.1.1 Opening a Terminal Device File

There is no additional rationale provided for this section.

A.11.1.2 Process Groups

There is a potential race when the members of the foreground process group on a terminal leave that process group, either by exit or by changing process groups. After the last process exits the process group, but before the foreground process group ID of the terminal is changed (usually by a job-control shell), it would be possible for a new process to be created with its process ID equal to the terminal's foreground process group ID. That process might then become the process group leader and accidentally be placed into the foreground on a terminal that was not necessarily its controlling terminal. As a result of this problem, the controlling terminal is defined to not have a foreground process group during this time.

The cases where a controlling terminal has no foreground process group occur when all processes in the foreground process group either terminate and are waited for or join other process groups via *setpgid*( ) or *setsid*( ). If the process group leader terminates, this is the first case described; if it leaves the process group via *setpgid*( ), this is the second case described (a process group leader cannot successfully call *setsid*( )). When one of those cases causes a controlling terminal to have no foreground process group, it has two visible effects on applications. The first is the value returned by *tcgetpgrp*( ). The second (which occurs only in the case where the process group leader terminates) is the sending of signals in response to special input characters. The intent of IEEE Std. 1003.1-200x is that no process group be wrongly identified as the foreground process group by *tcgetpgrp*( ) or unintentionally receive signals because of placement into the foreground.

In 4.3 BSD, the old process group ID continues to be used to identify the foreground process group and is returned by the function equivalent to *tcgetpgrp*( ). In that implementation it is possible for a newly created process to be assigned the same value as a process ID and then form a new process group with the same value as a process group ID. The result is that the new process group would receive signals from this terminal for no apparent reason, and IEEE Std. 1003.1-200x precludes this by forbidding a process group from entering the foreground in this way. It would be more direct to place part of the requirement made by the last sentence under *fork*( ), but there is no convenient way for that section to refer to the value that *tcgetpgrp*( ) returns, since in this case there is no process group and thus no process group ID.

One possibility for a conforming implementation is to behave similarly to 4.3 BSD, but to prevent this reuse of the ID, probably in the implementation of *fork*( ), as long as it is in use by the terminal.

Another possibility is to recognize when the last process stops using the terminal's foreground process group ID, which is when the process group lifetime ends, and to change the terminal's foreground process group ID to a reserved value that is never used as a process ID or process group ID. (See the definition of *process group lifetime* in the definitions section.) The process ID can then be reserved until the terminal has another foreground process group.

The 4.3 BSD implementation permits the leader (and only member) of the foreground process group to leave the process group by calling the equivalent of *setpgid*( ) and to later return, expecting to return to the foreground. There are no known application needs for this behavior, and IEEE Std. 1003.1-200x neither requires nor forbids it (except that it is forbidden for session leaders) by leaving it unspecified.

A.11.1.3 The Controlling Terminal

IEEE Std. 1003.1-200x does not specify a mechanism by which to allocate a controlling terminal. This is normally done by a system utility (such as *getty*) and is considered an administrative feature outside the scope of IEEE Std. 1003.1-200x.

Historical implementations allocate controlling terminals on certain *open*( ) calls. Since *open*( ) is part of POSIX.1, its behavior had to be dealt with. The traditional behavior is not required because it is not very straightforward or flexible for either implementations or applications. However, because of its prevalence, it was not practical to disallow this behavior either. Thus, a mechanism was standardized to ensure portable, predictable behavior in *open*( ).

Some historical implementations deallocate a controlling terminal on the last system-wide close. This behavior in neither required nor prohibited. Even on implementations that do provide this behavior, applications generally cannot depend on it due to its system-wide nature.

A.11.1.4 Terminal Access Control

The access controls described in this section apply only to a process that is accessing its controlling terminal. A process accessing a terminal that is not its controlling terminal is effectively treated the same as a member of the foreground process group. While this may seem unintuitive, note that these controls are for the purpose of job control, not security, and job control relates only to a process' controlling terminal. Normal file access permissions handle security.

If the process calling *read*( ) or *write*( ) is in a background process group that is orphaned, it is not desirable to stop the process group, as it is no longer under the control of a job control shell that could put it into foreground again. Accordingly, calls to *read*( ) or *write*( ) functions by such processes receive an immediate error return. This is different than in 4.2 BSD, which kills orphaned processes that receive terminal stop signals.

The foreground / background / orphaned process group check performed by the terminal driver must be repeatedly performed until the calling process moves into the foreground or until the process group of the calling process becomes orphaned. That is, when the terminal driver determines that the calling process is in the background and should receive a job control signal, it sends the appropriate signal (SIGTTIN or SIGTTOU) to every process in the process group of the calling process and then it allows the calling process to immediately receive the signal. The latter is typically performed by blocking the process so that the signal is immediately noticed. Note, however, that after the process finishes receiving the signal and control is returned to the driver, the terminal driver must reexecute the foreground / background / orphaned process group check. The process may still be in the background, either because it was continued in the background by a job-control shell, or because it caught the signal and did nothing.

The terminal driver repeatedly performs the foreground / background / orphaned process group checks whenever a process is about to access the terminal. In the case of *write*( ) or the control *tc**( ) functions, the check is performed at the entry of the function. In the case of *read*( ), the check is performed not only at the entry of the function, but also after blocking the process to wait for input characters (if necessary). That is, once the driver has determined that the process calling the *read*( ) function is in the foreground, it attempts to retrieve characters from the input queue. If the queue is empty, it blocks the process waiting for characters. When characters are available and control is returned to the driver, the terminal driver must return to the repeated foreground / background / orphaned process group check again. The process may have moved from the foreground to the background while it was blocked waiting for input characters.

A.11.1.5 Input Processing and Reading Data

There is no additional rationale provided for this section.

A.11.1.6 Canonical Mode Input Processing

The term *character* is intended here. ERASE should erase the last character, not the last byte. In the case of multi-byte characters, these two may be different.

4.3 BSD has a WERASE character that erases the last ''word'' typed (but not any preceding <blank>s or <tab>s). A word is defined as a sequence of non-<blank> characters, with <tab>s counted as <blank>s. Like ERASE, WERASE does not erase beyond the beginning of the line. This WERASE feature has not been specified in POSIX.1 because it is difficult to define in the international environment. It is only useful for languages where words are delimited by <blank>s. In some ideographic languages, such as Japanese and Chinese, words are not delimited at all. The WERASE character should presumably take one back to the beginning of a sentence in those cases; practically, this means it would not get much use for those languages.

It should be noted that there is a possible inherent deadlock if the application and implementation conflict on the value of MAX_CANON. With ICANON set (if IXOFF is enabled) and more than MAX_CANON characters transmitted without a <linefeed>, transmission will be stopped, the <linefeed> (or <carriage-return> when ICRLF is set) will never arrive, and the *read*( ) will never be satisfied.

An application should not set IXOFF if it is using canonical mode unless it knows that (even in the face of a transmission error) the conditions described previously cannot be met or unless it is prepared to deal with the possible deadlock in some other way, such as timeouts.

It should also be noted that this can be made to happen in non-canonical mode if the trigger value for sending IXOFF is less than VMIN and VTIME is zero.

A.11.1.7 Non-Canonical Mode Input Processing

Some points to note about MIN and TIME:

1. The interactions of MIN and TIME are not symmetric. For example, when MIN>0 and TIME=0, TIME has no effect. However, in the opposite case where MIN=0 and TIME>0, both MIN and TIME play a role in that MIN is satisfied with the receipt of a single character.

2. Also note that in case A (MIN>0, TIME>0), TIME represents an inter-character timer, while in case C (MIN=0, TIME>0), TIME represents a read timer.

These two points highlight the dual purpose of the MIN / TIME feature. Cases A and B, where MIN>0, exist to handle burst-mode activity (for example, file transfer programs) where a program would like to process at least MIN characters at a time. In case A, the inter-character timer is activated by a user as a safety measure; in case B, it is turned off.

Cases C and D exist to handle single-character timed transfers. These cases are readily adaptable to screen-based applications that need to know if a character is present in the input queue before refreshing the screen. In case C, the read is timed; in case D, it is not.

Another important note is that MIN is always just a minimum. It does not denote a record length. That is, if a program does a read of 20 bytes, MIN is 10, and 25 characters are present, 20 characters shall be returned to the user. In the special case of MIN=0, this still applies: if more than one character is available, they all will be returned immediately.

A.11.1.8 Writing Data and Output Processing

There is no additional rationale for this section.

A.11.1.9 Special Characters

There is no additional rationale for this section.

A.11.1.10 Modem Disconnect

There is no additional rationale for this section.

A.11.1.11 Closing a Terminal Device File

IEEE Std. 1003.1-200x does not specify that a *close*( ) on a terminal device file include the equivalent of a call to *tcflow*(*fd*,TCOON).

An implementation that discards output at the time *close*( ) is called after reporting the return value to the *write*( ) call that data was written does not conform with IEEE Std. 1003.1-200x. An application has functions such as *tcdrain*( ), *tcflush*( ), and *tcflow*( ) available to obtain the detailed behavior it requires with respect to flushing of output.

At the time of the last close on a terminal device, an application relinquishes any ability to exert flow control via *tcflow*( ).

A.11.2 Parameters that Can be Set

A.11.2.1 The termios Structure

This structure is part of an interface that, in general, retains the historic grouping of flags. Although a more optimal structure for implementations may be possible, the degree of change to applications would be significantly larger.

A.11.2.2 Input Modes

Some historical implementations treated a long break as multiple events, as many as one per character time. The wording in POSIX.1 explicitly prohibits this.

Although the ISTRIP flag is normally superfluous with today's terminal hardware and software, it is historically supported. Therefore, applications may be using ISTRIP, and there is no technical problem with supporting this flag. Also, applications may wish to receive only 7-bit input bytes and may not be connected directly to the hardware terminal device (for example, when a connection traverses a network).

Also, there is no requirement in general that the terminal device ensures that high-order bits beyond the specified character size are cleared. ISTRIP provides this function for 7-bit characters, which are common.

In dealing with multi-byte characters, the consequences of a parity error in such a character, or in an escape sequence affecting the current character set, are beyond the scope of POSIX.1 and are best dealt with by the application processing the multi-byte characters.

A.11.2.3 Output Modes

POSIX.1 does not describe postprocessing of output to a terminal or detailed control of that from a portable application. (That is, translation of <newline> to <carriage-return> followed by <linefeed> or <tab> processing.) There is nothing that a portable application should do to its output for a terminal because that would require knowledge of the operation of the terminal. It is the responsibility of the operating system to provide postprocessing appropriate to the output device, whether it is a terminal or some other type of device.

Extensions to POSIX.1 to control the type of postprocessing already exist and are expected to continue into the future. The control of these features is primarily to adjust the interface between the system and the terminal device so the output appears on the display correctly. This should be set up before use by any application.

In general, both the input and output modes should not be set absolutely, but rather modified from the inherited state.

A.11.2.4 Control Modes

This section could be misread that the symbol ''CSIZE'' is a title in the **termios** *c_cflag* field . Although it does serve that function, it is also a required symbol, as a literal reading of POSIX.1 (and the caveats about typography) would indicate.

A.11.2.5 Local Modes

Non-canonical mode is provided to allow fast bursts of input to be read efficiently while still allowing single-character input.

The ECHONL function historically has been in many implementations. Since there seems to be no technical problem with supporting ECHONL, it is included in POSIX.1 to increase consensus.

The alternate behavior possible when ECHOK or ECHOE are specified with ICANON is permitted as a compromise depending on what the actual terminal hardware can do. Erasing characters and lines is preferred, but is not always possible.

A.11.2.6 Special Control Characters

Permitting VMIN and VTIME to overlap with VEOF and VEOL was a compromise for historical implementations. Only when backwards-compatibility of object code is a serious concern to an implementor should an implementation continue this practice. Correct applications that work with the overlap (at the source level) should also work if it is not present, but not the reverse.

A.12 Utility Conventions

A.12.1 Utility Argument Syntax

The standard developers considered that recent trends toward diluting the SYNOPSIS sections of historical reference pages to the equivalent of:

```
command [options][operands]
```

were a disservice to the reader. Therefore, considerable effort was placed into rigorous definitions of all the command line arguments and their interrelationships. The relationships depicted in the synopses are normative parts of IEEE Std. 1003.1-200x; this information is sometimes repeated in textual form, but that is only for clarity within context.

The use of ''undefined'' for conflicting argument usage and for repeated usage of the same option is meant to prevent portable applications from using conflicting arguments or repeated options unless specifically allowed (as is the case with *ls*, which allows simultaneous, repeated use of the **−C**, **−l**, and **−1** options). Many historical implementations will tolerate this usage, choosing either the first or the last applicable argument. This tolerance can continue, but portable applications cannot rely upon it. (Other implementations may choose to print usage messages instead.)

The use of ''undefined'' for conflicting argument usage also allows an implementation to make reasonable extensions to utilities where the implementor considers mutually-exclusive options according to IEEE Std. 1003.1-200x to have a sensible meaning and result.

IEEE Std. 1003.1-200x does not define the result of a command when an option-argument or operand is not followed by ellipses and the application specifies more than one of that option-argument or operand. This allows an implementation to define valid (although non-standard) behavior for the utility when more than one such option or operand is specified.

Allowing <blank> characters after an option (that is, placing an option and its option-argument into separate argument strings) when IEEE Std. 1003.1-200x does not require it encourages portability of users, while still preserving backwards-compatibility of scripts. Inserting <blank> characters between the option and the option-argument is preferred; however, historical usage has not been consistent in this area; therefore, <blank>s are required to be handled by all implementations, but implementations are also allowed to handle the historical syntax. Another justification for selecting the multiple-argument method was that the single-argument case is inherently ambiguous when the option-argument can legitimately be a null string.

IEEE Std. 1003.1-200x explicitly states that digits are permitted as operands and option-arguments. The lower and upper bounds for the values of the numbers used for operands and option-arguments were derived from the ISO C standard values for {LONG_MIN} and {LONG_MAX}. The requirement on the standard utilities is that numbers in the specified range do not cause a syntax error, although the specification of a number need not be semantically correct for a particular operand or option-argument of a utility. For example, the specification of:

```
dd obs=3000000000
```

would yield undefined behavior for the application and would be a syntax error because the number 3 000 000 000 is outside of the range −2 147 483 647 to +2 147 483 647. On the other hand:

```
dd obs=2000000000
```

may cause some error, such as ''blocksize too large'', rather than a syntax error.

A.12.2 Utility Syntax Guidelines

This section is based on the rules listed in the SVID. It was included for two reasons:

1. The individual utility descriptions in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Chapter 4, Utilities needed a set of common (although not universal) actions on which they could anchor their descriptions of option and operand syntax. Most of the standard utilities actually do use these guidelines, and many of their historical implementations use the *getopt*( ) function for their parsing. Therefore, it was simpler to cite the rules and merely identify exceptions.

2. Writers of portable applications need suggested guidelines if the POSIX community is to avoid the chaos of historical UNIX system command syntax.

It is recommended that all *future* utilities and applications use these guidelines to enhance ''user portability''. The fact that some historical utilities could not be changed (to avoid breaking historical applications) should not deter this future goal.

The voluntary nature of the guidelines is highlighted by repeated uses of the word *should* throughout. This usage should not be misinterpreted to imply that utilities that claim conformance in their OPTIONS sections do not always conform.

Guidelines 1 and 2 are offered as guidance for locales using Latin alphabets. No recommendations are made by IEEE Std. 1003.1-200x concerning utility naming in other locales.

In the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.9.1, Simple Commands, it is further stated that a command used in the Shell Command Language cannot be named with a trailing colon.

Guideline 3 was changed to allow alphanumeric characters (letters and digits) from the character set to allow compatibility with historical usage. Historical practice allows the use of digits wherever practical, and there are no portability issues that would prohibit the use of digits. In fact, from an internationalization viewpoint, digits (being non-language-dependent) are preferable over letters (a **–2** is intuitively self-explanatory to any user, while in the **–f** *filename* the letter `'f'` is a mnemonic aid only to speakers of Latin-based languages where ''file name'' happens to translate to a word that begins with `'f'`. Since guideline 3 still retains the word ''single'', multi-digit options are not allowed. Instances of historical utilities that used them have been marked obsolescent, with the numbers being changed from option names to option-arguments.

It was difficult to achieve a satisfactory solution to the problem of name space in option characters. When the standard developers desired to extend the historical *cc* utility to accept ISO C standard programs, they found that all of the portable alphabet was already in use by various vendors. Thus, they had to devise a new name, *c89*, rather than something like *cc* **–X**. There were suggestions that implementors be restricted to providing extensions through various means (such as using a plus sign as the option delimiter or using option characters outside the alphanumeric set) that would reserve all of the remaining alphanumeric characters for future POSIX standards. These approaches were resisted because they lacked the historical style of UNIX systems. Furthermore, if a vendor-provided option should become commonly used in the industry, it would be a candidate for standardization. It would be desirable to standardize such a feature using historical practice for the syntax (the semantics can be standardized with any syntax). This would not be possible if the syntax was one reserved for the vendor. However, since the standardization process may lead to minor changes in the semantics, it may prove to be better for a vendor to use a syntax that will not be affected by standardization.

Guideline 8 includes the concept of comma-separated lists in a single argument. It is up to the utility to parse such a list itself because *getopt*( ) just returns the single string. This situation was retained so that certain historical utilities would not violate the guidelines. Applications preparing for international use should be aware of an occasional problem with comma-separated lists: in some locales, the comma is used as the radix character. Thus, if an application is preparing operands for a utility that expects a comma-separated lists, it should avoid generating non-integer values through one of the means that is influenced by setting the *LC_NUMERIC* variable (such as *awk*, *bc*, *printf*, or *printf*( )).

Applications calling any utility with a first operand starting with `'−'` should usually specify −−, as indicated by Guideline 10, to mark the end of the options. This is true even if the SYNOPSIS in the Shell and Utilities volume of IEEE Std. 1003.1-200x does not specify any options; implementations may provide options as extensions to the Shell and Utilities volume of IEEE Std. 1003.1-200x. The standard utilities that do not support Guideline 10 indicate that fact in the OPTIONS section of the utility description.

Guideline 11 was modified to clarify that the order of different options should not matter relative to one another. However, the order of repeated options that also have option-arguments may be significant; therefore, such options are required to be interpreted in the order that they are specified. The *make* utility is an instance of a historical utility that uses repeated options in which the order is significant. Multiple files are specified by giving multiple instances of the −**f** option; for example:

```
make −f common_header −f specific_rules target
```

Guideline 13 does not imply that all of the standard utilities automatically accept the operand `'−'` to mean standard input or output, nor does it specify the actions of the utility upon encountering multiple `'−'` operands. It simply says that, by default, `'−'` operands are not used for other purposes in the file reading or writing (but not when using *stat*, *unlink*, *touch*, and so on) utilities. All information concerning actual treatment of the `'−'` operand is found in the individual utility sections.

An area of concern was that as implementations mature, implementation-defined utilities and implementation-defined utility options will result. The idea was expressed that there needed to be a standard way, say an environment variable or some such mechanism, to identify implementation-defined utilities separately from standard utilities that may have the same name. It was decided that there already exist several ways of dealing with this situation and that it is outside of the POSIX.2 scope to attempt to standardize in the area of non-standard items. A method that exists on some historical implementations is the use of the so-called **/local/bin** or **/usr/local/bin** directory to separate local or additional copies or versions of utilities. Another method that is also used is to isolate utilities into completely separate domains. Still another method to ensure that the desired utility is being used is to request the utility by its full path name. There are many approaches to this situation; the examples given above serve to illustrate that there is more than one.

A.13 Headers

A.13.1 Format of Entries

Each header reference page has a common layout of sections describing the interface. This layout is similar to the manual page or ‘‘man’’ page format shipped with most UNIX systems, and each header has sections describing the SYNOPSIS and DESCRIPTION. These are the two sections that relate to conformance.

Additional sections are informative, and add considerable information for the application developer. APPLICATION USAGE sections provide additional caveats, issues, and recommendations to the developer. RATIONALE sections give additional information on the decisions made in defining the interface.

FUTURE DIRECTIONS sections act as pointers to related work that may impact the interface in the future, and often cautions the developer to architect the code to account for a change in this area. Note that a future directions statement should not be taken as a commitment to adopt a feature or interface in the future.

The CHANGE HISTORY section describes when the interface was introduced, and how it has changed.

Option labels and margin markings in the page can be useful in guiding the application developer.

Rationale (Informative)

Part B:

System Interfaces

The Open Group

Appendix B

Rationale for System Interfaces

B.1 Introduction

B.1.1 Scope

Refer to Section A.1.1 (on page 3311).

B.1.2 Conformance

Refer to Section A.2 (on page 3317).

B.1.3 Normative References

There is no additional rationale for this section.

B.1.4 Changes from Issue 4

The change history is provided as an informative section, to track changes from previous issues of IEEE Std. 1003.1-200x that comprised earlier versions of the Single UNIX Specification.

B.1.4.1 Changes from Issue 4 to Issue 4, Version 2

There is no additional rationale for this section.

B.1.4.2 Changes from Issue 4, Version 2 to Issue 5

There is no additional rationale for this section.

B.1.4.3 Changes from Issue 5 to Issue 6 (IEEE Std. 1003.1-200x)

There is no additional rationale for this section.

B.1.5 New Features

There is no additional rationale for this section.

B.1.5.1 New Features in Issue 4, Version 2

There is no additional rationale for this section.

B.1.5.2 New Features in Issue 5

There is no additional rationale for this section.

B.1.5.3 New Features in Issue 6

There is no additional rationale for this section.

B.1.6 Terminology

Refer to Section A.1.4 (on page 3313).

B.1.7 Definitions

Refer to Section A.3 (on page 3321).

B.1.8 Relationship to Other Formal Standards

There is no additional rationale for this section.

B.1.9 Portability

Refer to Section A.1.5 (on page 3315).

B.1.9.1 Codes

Refer to Section A.1.5.1 (on page 3315).

B.1.10 Format of Entries

Each system interface reference page has a common layout of sections describing the interface. This layout is similar to the manual page or ''man'' page format shipped with most UNIX systems, and each header has sections describing the SYNOPSIS, DESCRIPTION, RETURN VALUE, and ERRORS. These are the four sections that relate to conformance.

Additional sections are informative, and add considerable information for the application developer. EXAMPLES sections provide example usage. APPLICATION USAGE sections provide additional caveats, issues, and recommendations to the developer. RATIONALE sections give additional information on the decisions made in defining the interface.

FUTURE DIRECTIONS sections act as pointers to related work that may impact the interface in the future, and often cautions the developer to architect the code to account for a change in this area. Note that a future directions statement should not be taken as a commitment to adopt a feature or interface in the future.

The CHANGE HISTORY section describes when the interface was introduced, and how it has changed.

Option labels and margin markings in the page can be useful in guiding the application developer.

B.2 General Information

B.2.1 Use and Implementation of Functions

The information concerning the use of functions was adapted from a description in the ISO C standard. Here is an example of how an application program can protect itself from library functions that may or may not be macros, rather than true functions:

The *atoi*( ) function may be used in any of several ways:

- By use of its associated header (possibly generating a macro expansion):

```
#include <stdlib.h>
/* ... */
i = atoi(str);
```

- By use of its associated header (assuredly generating a true function call):

```
#include <stdlib.h>
#undef atoi
/* ... */
i = atoi(str);
```

  or:

```
#include <stdlib.h>
/* ... */
i = (atoi) (str);
```

- By explicit declaration:

```
extern int atoi (const char *);
/* ... */
i = atoi(str);
```

- By implicit declaration:

```
/* ... */
i = atoi(str);
```

  (Assuming no function prototype is in scope. This is not allowed by the ISO C standard for functions with variable arguments; furthermore, parameter type conversion ‘‘widening’’ is subject to different rules in this case.)

Note that the ISO C standard reserves names starting with `'_'` for the compiler. Therefore, the compiler could, for example, implement an intrinsic, built-in function *_asm_builtin_atoi*( ), which it recognized and expanded into inline assembly code. Then, in **<stdlib.h>**, there could be the following:

```
#define atoi(X) _asm_builtin_atoi(X)
```

The user’s ‘‘normal’’ call to *atoi*( ) would then be expanded inline, but the implementor would also be required to provide a callable function named *atoi*( ) for use when the application requires it; for example, if its address is to be stored in a function pointer variable.

B.2.2 The Compilation Environment

B.2.2.1 POSIX.1 Symbols

This and the following section address the issue of ‘‘name space pollution’’. The ISO C standard requires that the name space beyond what it reserves not be altered except by explicit action of the application writer. This section defines the actions to add the POSIX.1 symbols for those headers where both the ISO C standard and POSIX.1 need to define symbols, and also where the XSI Extension extends the base standard.

When headers are used to provide symbols, there is a potential for introducing symbols that the application writer cannot predict. Ideally, each header should only contain one set of symbols, but this is not practical for historical reasons. Thus, the concept of feature test macros is included. Two feature test macros are explicitly defined by IEEE Std. 1003.1-200x; it is expected that future revisions may add to this.

It is further intended that these feature test macros apply only to the headers specified by IEEE Std. 1003.1-200x. Implementations are expressly permitted to make visible symbols not specified by IEEE Std. 1003.1-200x, within both POSIX.1 and other headers, under the control of feature test macros that are not defined by IEEE Std. 1003.1-200x.

The _POSIX_C_SOURCE Feature Test Macro

Since _POSIX_SOURCE specified by the POSIX.1-1990 standard did not have a value associated with it, the _POSIX_C_SOURCE macro replaces it, allowing an application to inform the system of the revision of the standard to which it conforms. This symbol will allow implementations to support various revisions of IEEE Std. 1003.1-200x simultaneously. For instance, when either _POSIX_SOURCE is defined or _POSIX_C_SOURCE is defined as 1, the system should make visible the same name space as permitted and required by the POSIX.1-1990 standard. When _POSIX_C_SOURCE is defined, the state of _POSIX_SOURCE is completely irrelevant.

It is expected that C bindings to future POSIX standards will define new values for _POSIX_C_SOURCE, with each new value reserving the name space for that new standard, plus all earlier POSIX standards. Using a single feature test macro for all standards rather than a separate macro for each standard furthers the goal of eventually combining all of the C bindings into one standard.

It is further intended that these feature test macros apply only to the headers specified by IEEE Std. 1003.1-200x. Implementations are expressly permitted to make visible symbols not specified by IEEE Std. 1003.1-200x, within both IEEE Std. 1003.1-200x and other headers, under the control of feature test macros that are not defined by IEEE Std. 1003.1-200x.

B.2.2.2 The Name Space

The reservation of identifiers is paraphrased from the ISO C standard. The text is included because it needs to be part of IEEE Std. 1003.1-200x, regardless of possible changes in future versions of the ISO C standard.

These identifiers may be used by implementations, particularly for feature test macros. Implementations should not use feature test macro names that might be reasonably used by a standard.

Including headers more than once is a reasonably common practice, and it should be carried forward from the ISO C standard. More significantly, having definitions in more than one header is explicitly permitted. Where the potential declaration is ‘‘benign’’ (the same definition twice) the declaration can be repeated, if that is permitted by the compiler. (This is usually true of macros, for example.) In those situations where a repetition is not benign (for example,

typedefs), conditional compilation must be used. The situation actually occurs both within the ISO C standard and within POSIX.1: **time_t** should be in **<sys/types.h>**, and the ISO C standard mandates that it be in **<time.h>**.

The area of name space pollution *versus* additions to structures is difficult because of the macro structure of C. The following discussion summarizes all the various problems with and objections to the issue.

Note the phrase ''user-defined macro''. Users are not permitted to define macro names (or any other name) beginning with `"_[A-Z_]"`. Thus, the conflict cannot occur for symbols reserved to the vendor's name space, and the permission to add fields automatically applies, without qualification, to those symbols.

1. Data structures (and unions) need to be defined in headers by implementations to meet certain requirements of POSIX.1 and the ISO C standard.

2. The structures defined by POSIX.1 are typically minimal, and any practical implementation would wish to add fields to these structures either to hold additional related information or for backwards-compatibility (or both). Future standards (and *de facto* standards) would also wish to add to these structures. Issues of field alignment make it impractical (at least in the general case) to simply omit fields when they are not defined by the particular standard involved.

   Struct **dirent** is an example of such a minimal structure (although one could argue about whether the other fields need visible names). The *st_rdev* field of most implementations' **stat** structure is a common example where extension is needed and where a conflict could occur.

3. Fields in structures are in an independent name space, so the addition of such fields presents no problem to the C language itself in that such names cannot interact with identically named user symbols because access is qualified by the specific structure name.

4. There is an exception to this: macro processing is done at a lexical level. Thus, symbols added to a structure might be recognized as user-provided macro names at the location where the structure is declared. This only can occur if the user-provided name is declared as a macro before the header declaring the structure is included. The user's use of the name after the declaration cannot interfere with the structure because the symbol is hidden and only accessible through access to the structure. Presumably, the user would not declare such a macro if there was an intention to use that field name.

5. Macros from the same or a related header might use the additional fields in the structure, and those field names might also collide with user macros. Although this is a less frequent occurrence, since macros are expanded at the point of use, no constraint on the order of use of names can apply.

6. An ''obvious'' solution of using names in the reserved name space and then redefining them as macros when they should be visible does not work because this has the effect of exporting the symbol into the general name space. For example, given a (hypothetical) system-provided header **<h.h>**, and two parts of a C program in **a.c** and **b.c**, in header **<h.h>**:

```
struct foo {
    int __i;
}
```

```
#ifdef _FEATURE_TEST
#define i __i;
#endif
```

In file **a.c**:

```
#include h.h
extern int i;
...
```

In file **b.c**:

```
extern int i;
...
```

The symbol that the user thinks of as *i* in both files has an external name of *__i* in **a.c**; the same symbol *i* in **b.c** has an external name *i* (ignoring any hidden manipulations the compiler might perform on the names). This would cause a mysterious name resolution problem when **a.o** and **b.o** are linked.

Simply avoiding definition then causes alignment problems in the structure.

A structure of the form:

```
struct foo {
    union {
        int __i;
#ifdef _FEATURE_TEST
        int i;
#endif
    } __ii;
}
```

does not work because the name of the logical field *i* is *__ii.i*, and introduction of a macro to restore the logical name immediately reintroduces the problem discussed previously (although its manifestation might be more immediate because a syntax error would result if a recursive macro did not cause it to fail first).

7. A more workable solution would be to declare the structure:

```
struct foo {
#ifdef _FEATURE_TEST
    int i;
#else
    int __i;
#endif
}
```

However, if a macro (particularly one required by a standard) is to be defined that uses this field, two must be defined: one that uses *i*, the other that uses *__i*. If more than one additional field is used in a macro and they are conditional on distinct combinations of features, the complexity goes up as 2^n.

All this leaves a difficult situation: vendors must provide very complex headers to deal with what is conceptually simple and safe—adding a field to a structure. It is the possibility of user-provided macros with the same name that makes this difficult.

Several alternatives were proposed that involved constraining the user's access to part of the name space available to the user (as specified by the ISO C standard). In some cases, this was only until all the headers had been included. There were two proposals discussed that failed to achieve consensus:

1. Limiting it for the whole program.
2. Restricting the use of identifiers containing only uppercase letters until after all system headers had been included. It was also pointed out that because macros might wish to access fields of a structure (and macro expansion occurs totally at point of use) restricting names in this way would not protect the macro expansion, and thus the solution was inadequate.

It was finally decided that reservation of symbols would occur, but as constrained.

The current wording also allows the addition of fields to a structure, but requires that user macros of the same name not interfere. This allows vendors to do one of the following:

- Not create the situation (do not extend the structures with user-accessible names or use the solution in (7) above)
- Extend their compilers to allow some way of adding names to structures and macros safely

There are at least two ways that the compiler might be extended: add new preprocessor directives that turn off and on macro expansion for certain symbols (without changing the value of the macro) and a function or lexical operation that suppresses expansion of a word. The latter seems more flexible, particularly because it addresses the problem in macros as well as in declarations.

The following seems to be a possible implementation extension to the C language that will do this: any token that during macro expansion is found to be preceded by three '#' symbols shall not be further expanded in exactly the same way as described for macros that expand to their own name as in Section 3.8.3.4 of the ISO C standard. A vendor may also wish to implement this as an operation that is lexically a function, which might be implemented as:

```
#define __safe_name(x) ###x
```

Using a function notation would insulate vendors from changes in standards until such a functionality is standardized (if ever). Standardization of such a function would be valuable because it would then permit third parties to take advantage of it portably in software they may supply.

The symbols that are ''explicitly permitted, but not required by IEEE Std. 1003.1-200x'' include those classified below. (That is, the symbols classified below might, but are not required to, be present when _POSIX_C_SOURCE is defined to have the value 20010xL.)

- Symbols in **<limits.h>** and **<unistd.h>** that are defined to indicate support for options or limits that are constant at compile-time.
- Symbols in the name space reserved for the implementation by the ISO C standard.
- Symbols in a name space reserved for a particular type of extension (for example, type names ending with **_t** in **<sys/types.h>**).
- Additional members of structures or unions whose names do not reduce the name space reserved for applications.

Since both implementations and future revisions of IEEE Std. 1003.1-200x and other POSIX standards may use symbols in the reserved spaces described in these tables, there is a potential for name space clashes. To avoid future name space clashes when adding symbols, implementations should not use the posix_, POSIX_, or _POSIX_ prefixes.

B.2.3 Error Numbers

It was the consensus of the standard developers that to allow the conformance document to state that an error occurs and under what conditions, but to disallow a statement that it never occurs, does not make sense. It could be implied by the current wording that this is allowed, but to reduce the possibility of future interpretation requests, it is better to make an explicit statement.

The ISO C standard requires that *errno* be an assignable *lvalue*. Originally, the definition in POSIX.1 was stricter than that in the ISO C standard, **extern int** *errno*, in order to support historical usage. In a multi-threaded environment, implementing *errno* as a global variable results in non-deterministic results when accessed. It is required, however, that *errno* work as a per-thread error reporting mechanism. In order to do this, a separate *errno* value has to be maintained for each thread. The following section discusses the various alternative solutions that were considered.

In order to avoid this problem altogether for new functions, these functions avoid using *errno* and, instead, return the error number directly as the function return value; a return value of zero indicates that no error was detected.

For any function that can return errors, the function return value is not used for any purpose other than for reporting errors. Even when the output of the function is scalar, it is passed through a function argument. While it might have been possible to allow some scalar outputs to be coded as negative function return values and mixed in with positive error status returns, this was rejected—using the return value for a mixed purpose was judged to be of limited use and error prone.

Checking the value of *errno* alone is not sufficient to determine the existence or type of an error, since it is not required that a successful function call clear *errno*. The variable *errno* should only be examined when the return value of a function indicates that the value of *errno* is meaningful. In that case, the function is required to set the variable to something other than zero.

The variable *errno* shall never be set to zero by any function call; to do so would contradict the ISO C standard.

POSIX.1 requires (in the ERRORS sections of function descriptions) certain error values to be set in certain conditions because many existing applications depend on them. Some error numbers, such as [EFAULT], are entirely implementation-defined and are noted as such in their description in the ERRORS section. This section otherwise allows wide latitude to the implementation in handling error reporting.

Some of the ERRORS sections in IEEE Std. 1003.1-200x have two subsections. The first:

> ‘‘The function shall fail if:’’

could be called the ‘‘mandatory’’ section.

The second:

> ‘‘The function may fail if:’’

could be informally known as the ‘‘optional’’ section.

Attempting to infer the quality of an implementation based on whether it detects optional error conditions is not useful.

Following each one-word symbolic name for an error, there is a description of the error. The rationale for some of the symbolic names follows:

[ECANCELED] This spelling was chosen as being more common.

[EFAULT] Most historical implementations do not catch an error and set *errno* when an invalid address is given to the functions *wait*( ), *time*( ), or *times*( ). Some implementations cannot reliably detect an invalid address. And most systems that detect invalid addresses will do so only for a system call, not for a library routine.

[EFTYPE] This error code was proposed in earlier proposals as ‘‘Inappropriate operation for file type’’, meaning that the operation requested is not appropriate for the file specified in the function call. This code was proposed, although the same idea was covered by [ENOTTY], because the connotations of the name would be misleading. It was pointed out that the *fcntl*( ) function uses the error code [EINVAL] for this notion, and hence all instances of [EFTYPE] were changed to this code.

[EINTR] POSIX.1 prohibits conforming implementations from restarting interrupted system calls. However, it does not require that [EINTR] be returned when another legitimate value may be substituted; for example, a partial transfer count when *read*( ) or *write*( ) are interrupted. This is only given when the signal catching function returns normally as opposed to returns by mechanisms like *longjmp*( ) or *siglongjmp*( ).

[ELOOP] In specifying conditions under which implementations would generate this error, the following goals were considered:

- To ensure that actual loops are detected, including loops that result from symbolic links across distributed file systems.
- To ensure that during path name resolution an application can rely on the ability to follow at least {SYMLOOP_MAX} symbolic links in the absence of a loop.
- To allow implementations to provide the capability of traversing more than {SYMLOOP_MAX} symbolic links in the absence of a loop.
- To allow implementations to detect loops and generate the error prior to encountering {SYMLOOP_MAX} symbolic links.

[ENAMETOOLONG]
When a symbolic link is encountered during path name resolution, the contents of that symbolic link are used to create a new path name. The standard developers intended to allow, but not require, that implementations enforce the restriction of {PATH_MAX} on the result of this path name substitution.

[ENOMEM] The term *main memory* is not used in POSIX.1 because it is implementation-defined.

[ENOTSUP] This error code is to be used when an implementation chooses to implement the required functionality of IEEE Std. 1003.1-200x but does not support optional facilities defined by IEEE Std. 1003.1-200x. The return of [ENOSYS] is to be taken to indicate that the function of the interface is not supported at all; the function will always fail with this error code.

[ENOTTY] The symbolic name for this error is derived from a time when device control was done by *ioctl*( ) and that operation was only permitted on a terminal interface. The term *TTY* is derived from *teletypewriter*, the devices to which this error originally applied.

[EPIPE] This condition normally generates the signal SIGPIPE; the error is returned if the signal does not terminate the process.

[EROFS] In historical implementations, attempting to *unlink*( ) or *rmdir*( ) a mount point would generate an [EBUSY] error. An implementation could be envisioned where such an operation could be performed without error. In this case, if *either* the directory entry or the actual data structures reside on a read-only file system, [EROFS] is the appropriate error to generate. (For example, changing the link count of a file on a read-only file system could not be done, as is required by *unlink*( ), and thus an error should be reported.)

Three error numbers, [EDOM], [EILSEQ], and [ERANGE], were added to this section primarily for consistency with the ISO C standard.

Alternative Solutions for Per-Thread errno

The usual implementation of *errno* as a single global variable does not work in a multi-threaded environment. In such an environment, a thread may make a POSIX.1 call and get a −1 error return, but before that thread can check the value of *errno*, another thread might have made a second POSIX.1 call that also set *errno*. This behavior is unacceptable in robust programs. There were a number of alternatives that were considered for handling the *errno* problem:

- Implement *errno* as a per-thread integer variable.
- Implement *errno* as a service that can access the per-thread error number.
- Change all POSIX.1 calls to accept an extra status argument and avoid setting *errno*.
- Change all POSIX.1 calls to raise a language exception.

The first option offers the highest level of compatibility with existing practice but requires special support in the linker, compiler, and/or virtual memory system to support the new concept of thread private variables. When compared with current practice, the third and fourth options are much cleaner, more efficient, and encourage a more robust programming style, but they require new versions of all of the POSIX.1 functions that might detect an error. The second option offers compatibility with existing code that uses the **<errno.h>** header to define the symbol *errno*. In this option, *errno* may be a macro defined:

```
#define errno  (*__errno())
extern int      *__errno();
```

This option may be implemented as a per-thread variable whereby an *errno* field is allocated in the user space object representing a thread, and whereby the function *__errno*( ) makes a system call to determine the location of its user space object and returns the address of the *errno* field of that object. Another implementation, one that avoids calling the kernel, involves allocating stacks in chunks. The stack allocator keeps a side table indexed by chunk number containing a pointer to the thread object that uses that chunk. The *__errno*( ) function then looks at the stack pointer, determines the chunk number, and uses that as an index into the chunk table to find its thread object and thus its private value of *errno*. On most architectures, this can be done in four to five instructions. Some compilers may wish to implement *__errno*( ) inline to improve performance.

Disallowing Return of the [EINTR] Error Code

Many blocking interfaces defined by IEEE Std. 1003.1-200x may return [EINTR] if interrupted during their execution by a signal handler. Blocking interfaces introduced under the Threads option do not have this property. Instead, they require that the interface appear to be atomic with respect to interruption. In particular, clients of block interfaces need not handle any possible [EINTR] return as a special case since it will never occur. If it is necessary to restart operations or complete incomplete operations following the execution of a signal handler, this is handled by the implementation, rather than by the application.

Requiring applications to handle [EINTR] errors on blocking interfaces has been shown to be a frequent source of often unreproducible bugs, and it adds no compelling value to the available functionality. Thus, blocking interfaces introduced for use by multi-threaded programs do not use this paradigm. In particular, in none of the functions *flockfile*( ), *pthread_cond_timedwait*( ), *pthread_cond_wait*( ), *pthread_join*( ), *pthread_mutex_lock*( ), and *sigwait*( ) did providing [EINTR] returns add value, or even particularly make sense. Thus, these functions do not provide for an [EINTR] return, even when interrupted by a signal handler. The same arguments can be applied to *sem_wait*( ), *sem_trywait*( ), *sigwaitinfo*( ), and *sigtimedwait*( ), but implementations are permitted to return [EINTR] error codes for these functions for compatibility with earlier versions of IEEE Std. 1003.1-200x. Applications cannot rely on calls to these functions returning [EINTR] error codes when signals are delivered to the calling thread, but they should allow for the possibility.

B.2.3.1 Additional Error Numbers

The ISO C standard defines the name space for implementations to add additional error numbers.

B.2.4 Signal Concepts

Historical implementations of signals, using the *signal*( ) function, have shortcomings that make them unreliable for many application uses. Because of this, a new signal mechanism, based very closely on the one of 4.2 BSD and 4.3 BSD, was added to POSIX.1.

Signal Names

The restriction on the actual type used for **sigset_t** is intended to guarantee that these objects can always be assigned, have their address taken, and be passed as parameters by value. It is not intended that this type be a structure including pointers to other data structures, as that could impact the portability of applications performing such operations. A reasonable implementation could be a structure containing an array of some integer type.

The signals described in IEEE Std. 1003.1-200x must have unique values so that they may be named as parameters of **case** statements in the body of a C language **switch** clause. However, implementation-defined signals may have values that overlap with each other or with signals specified in IEEE Std. 1003.1-200x. An example of this is SIGABRT, which traditionally overlaps some other signal, such as SIGIOT.

SIGKILL, SIGTERM, SIGUSR1, and SIGUSR2 are ordinarily generated only through the explicit use of the *kill*( ) function, although some implementations generate SIGKILL under extraordinary circumstances. SIGTERM is traditionally the default signal sent by the *kill* command.

The signals SIGBUS, SIGEMT, SIGIOT, SIGTRAP, and SIGSYS were omitted from POSIX.1 because their behavior is implementation-defined and could not be adequately categorized. Conforming implementations may deliver these signals, but must document the circumstances under which they are delivered and note any restrictions concerning their delivery. The signals SIGFPE, SIGILL, and SIGSEGV are similar in that they also generally result only from programming errors. They were included in POSIX.1 because they do indicate three relatively well-categorized conditions. They are all defined by the ISO C standard and thus would have to be defined by any system with a ISO C standard binding, even if not explicitly included in POSIX.1.

There is very little that a Conforming POSIX.1 Application can do by catching, ignoring, or masking any of the signals SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGBUS, SIGSEGV, SIGSYS, or SIGFPE. They will generally be generated by the system only in cases of programming errors. While it may be desirable for some robust code (for example, a library routine) to be able to detect and recover from programming errors in other code, these signals are not nearly sufficient for that purpose. One portable use that does exist for these signals is that a command interpreter can recognize them as the cause of a process' termination (with *wait*( )) and print an appropriate message. The mnemonic tags for these signals are derived from their PDP-11 origin.

The signals SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, and SIGCONT are provided for job control and are unchanged from 4.2 BSD. The signal SIGCHLD is also typically used by job control shells to detect children that have terminated or, as in 4.2 BSD, stopped.

Some implementations, including System V, have a signal named SIGCLD, which is similar to SIGCHLD in 4.2 BSD. POSIX.1 permits implementations to have a single signal with both names. POSIX.1 carefully specifies ways in which portable applications can avoid the semantic differences between the two different implementations. The name SIGCHLD was chosen for POSIX.1 because most current application usages of it can remain unchanged in conforming applications. SIGCLD in System V has more cases of semantics that POSIX.1 does not specify, and thus applications using it are more likely to require changes in addition to the name change.

The signals SIGUSR1 and SIGUSR2 are commonly used by applications for notification of exceptional behavior and are described as ''reserved as application-defined'' so that such use is not prohibited. Implementations should not generate SIGUSR1 or SIGUSR2, except when explicitly requested by *kill*( ). It is recommended that libraries not use these two signals, as such use in libraries could interfere with their use by applications calling the libraries. If such use is unavoidable, it should be documented. It is prudent for non-portable libraries to use non-standard signals to avoid conflicts with use of standard signals by portable libraries.

There is no portable way for an application to catch or ignore non-standard signals. Some implementations define the range of signal numbers, so applications can install signal-catching functions for all of them. Unfortunately, implementation-defined signals often cause problems when caught or ignored by applications that do not understand the reason for the signal. While the desire exists for an application to be more robust by handling all possible signals (even those only generated by *kill*( )), no existing mechanism was found to be sufficiently portable to include in POSIX.1. The value of such a mechanism, if included, would be diminished given that SIGKILL would still not be catchable.

A number of new signal numbers are reserved for applications because the two user signals defined by POSIX.1 are insufficient for many realtime applications. A range of signal numbers is specified, rather than an enumeration of additional reserved signal names, because different applications and application profiles will require a different number of application signals. It is not desirable to burden all application domains and therefore all implementations with the maximum number of signals required by all possible applications. Note that in this context, signal numbers are essentially different signal priorities.

The relatively small number of required additional signals, {_POSIX_RTSIG_MAX}, was chosen so as not to require an unreasonably large signal mask/set. While this number of signals defined in POSIX.1 will fit in a single 32-bit word signal mask, it is recognized that most existing implementations define many more signals than are specified in POSIX.1 and, in fact, many implementations have already exceeded 32 signals (including the ''null signal''). Support of {_POSIX_RTSIG_MAX} additional signals may push some implementation over the single 32-bit word line, but is unlikely to push any implementations that are already over that line beyond the 64-signal line.

B.2.4.1 Signal Generation and Delivery

The terms defined in this section are not used consistently in documentation of historical systems. Each signal can be considered to have a lifetime beginning with *generation* and ending with *delivery* or *acceptance*. The POSIX.1 definition of *delivery* does not exclude ignored signals; this is considered a more consistent definition. This revised text in several parts of IEEE Std. 1003.1-200x clarifies the distinct semantics of asynchronous signal *delivery* and synchronous signal *acceptance*. The previous wording attempted to categorize both under the term *delivery*, which led to conflicts over whether the effects of asynchronous signal delivery applied to synchronous signal acceptance.

Signals generated for a process are delivered to only one thread. Thus, if more than one thread is eligible to receive a signal, one has to be chosen. The choice of threads is left entirely up to the implementation both to allow the widest possible range of conforming implementations and to give implementations the freedom to deliver the signal to the ''easiest possible'' thread should there be differences in ease of delivery between different threads.

Note that should multiple delivery among cooperating threads be required by an application, this can be trivially constructed out of the provided single-delivery semantics. The construction of a *sigwait_multiple*( ) function that accomplishes this goal is presented with the rationale for *sigwaitinfo*( ).

Implementations should deliver unblocked signals as soon after they are generated as possible. However, it is difficult for POSIX.1 to make specific requirements about this, beyond those in *kill*( ) and *sigprocmask*( ). Even on systems with prompt delivery, scheduling of higher priority processes is always likely to cause delays.

In general, the interval between the generation and delivery of unblocked signals cannot be detected by an application. Thus, references to pending signals generally apply to blocked, pending signals. An implementation registers a signal as pending on the process when no thread has the signal unblocked and there are no threads blocked in a *sigwait*( ) function for that signal. Thereafter, the implementation delivers the signal to the first thread that unblocks the signal or calls a *sigwait*( ) function on a signal set containing this signal rather than choosing the recipient thread at the time the signal is sent.

In the 4.3 BSD system, signals that are blocked and set to SIG_IGN are discarded immediately upon generation. For a signal that is ignored as its default action, if the action is SIG_DFL and the signal is blocked, a generated signal remains pending. In the 4.1 BSD system and in System V, Release 3, two other implementations that support a somewhat similar signal mechanism, all ignored, blocked signals remain pending if generated. Because it is not normally useful for an application to simultaneously ignore and block the same signal, it was unnecessary for POSIX.1 to specify behavior that would invalidate any of the historical implementations.

There is one case in some historical implementations where an unblocked, pending signal does not remain pending until it is delivered. In the System V implementation of *signal*( ), pending signals are discarded when the action is set to SIG_DFL or a signal-catching routine (as well as to SIG_IGN). Except in the case of setting SIGCHLD to SIG_DFL, implementations that do this do not conform completely to POSIX.1. Some earlier proposals for POSIX.1 explicitly stated this, but these statements were redundant due to the requirement that functions defined by POSIX.1 not change attributes of processes defined by POSIX.1 except as explicitly stated.

POSIX.1 specifically states that the order in which multiple, simultaneously pending signals are delivered is unspecified. This order has not been explicitly specified in historical implementations, but has remained quite consistent and been known to those familiar with the implementations. Thus, there have been cases where applications (usually system utilities) have been written with explicit or implicit dependencies on this order. Implementors and others porting existing applications may need to be aware of such dependencies.

When there are multiple pending signals that are not blocked, implementations should arrange for the delivery of all signals at once, if possible. Some implementations stack calls to all pending signal-catching routines, making it appear that each signal-catcher was interrupted by the next signal. In this case, the implementation should ensure that this stacking of signals does not violate the semantics of the signal masks established by *sigaction*( ). Other implementations process at most one signal when the operating system is entered, with remaining signals saved for later delivery. Although this practice is widespread, this behavior is neither standardized nor endorsed. In either case, implementations should attempt to deliver signals associated with the current state of the process (for example, SIGFPE) before other signals, if possible.

In 4.2 BSD and 4.3 BSD, it is not permissible to ignore or explicitly block SIGCONT, because if blocking or ignoring this signal prevented it from continuing a stopped process, such a process could never be continued (only killed by SIGKILL). However, 4.2 BSD and 4.3 BSD do block SIGCONT during execution of its signal-catching function when it is caught, creating exactly this problem. A proposal was considered to disallow catching SIGCONT in addition to ignoring and blocking it, but this limitation led to objections. The consensus was to require that SIGCONT always continue a stopped process when generated. This removed the need to disallow ignoring or explicit blocking of the signal; note that SIG_IGN and SIG_DFL are equivalent for SIGCONT .

B.2.4.2 Realtime Signal Generation and Delivery

The Realtime Signals Extension option to POSIX.1 signal generation and delivery behavior is required for the following reasons:

- The **sigevent** structure is used by other POSIX.1 functions that result in asynchronous event notifications to specify the notification mechanism to use and other information needed by the notification mechanism. IEEE Std. 1003.1-200x defines only three symbolic values for the notification mechanism. SIGEV_NONE is used to indicate that no notification is required when the event occurs. This is useful for applications that use asynchronous I/O with polling for completion. SIGEV_SIGNAL indicates that a signal shall be generated when the event occurs. SIGEV_NOTIFY provides for ''callback functions'' for asynchronous notifications done by a function call within the context of a new thread. This provides a multi-threaded process a more natural means of notification than signals. The primary difficulty with previous notification approaches has been to specify the environment of the notification routine.

  — One approach is to limit the notification routine to call only functions permitted in a signal handler. While the list of permissible functions is clearly stated, this is overly restrictive.

  — A second approach is to define a new list of functions or classes of functions that are explicitly permitted or not permitted. This would give a programmer more lists to deal with, which would be awkward.

  — The third approach is to define completely the environment for execution of the notification function. A clear definition of an execution environment for notification is provided by executing the notification function in the environment of a newly created thread.

Implementations may support additional notification mechanisms by defining new values for *sigev_notify*.

For a notification type of SIGEV_SIGNAL, the other members of the **sigevent** structure defined by IEEE Std. 1003.1-200x specify the realtime signal—that is, the signal number and application-defined value that differentiates between occurrences of signals with the same number—that will be generated when the event occurs. The structure is defined in **<signal.h>**, even though the structure is not directly used by any of the signal functions, because it is part of the signals interface used by the POSIX.1b ''client functions''. When the client functions include **<signal.h>** to define the signal names, the **sigevent** structure will also be defined.

An application-defined value passed to the signal handler is used to differentiate between different ''events'' instead of requiring that the application use different signal numbers for several reasons:

— Realtime applications potentially handle a very large number of different events. Requiring that implementations support a correspondingly large number of distinct signal numbers will adversely impact the performance of signal delivery because the signal masks to be manipulated on entry and exit to the handlers will become large.

— Event notifications are prioritized by signal number (the rationale for this is explained in the following paragraphs) and the use of different signal numbers to differentiate between the different event notifications overloads the signal number more than has already been done. It also requires that the application writer make arbitrary assignments of priority to events that are logically of equal priority.

A union is defined for the application-defined value so that either an integer constant or a pointer can be portably passed to the signal-catching function. On some architectures a pointer cannot be cast to an **int** and *vice versa*.

Use of a structure here with an explicit notification type discriminant rather than explicit parameters to realtime functions, or embedded in other realtime structures, provides for future extensions to IEEE Std. 1003.1-200x. Additional, perhaps more efficient, notification mechanisms can be supported for existing realtime function interfaces, such as timers and asynchronous I/O, by extending the **sigevent** structure appropriately. The existing realtime function interfaces will not have to be modified to use any such new notification mechanism. The revised text concerning the SIGEV_SIGNAL value makes consistent the semantics of the members of the **sigevent** structure, particularly in the definitions of *lio_listio*( ) and *aio_fsync*( ). For uniformity, other revisions cause this specification to be referred to rather than inaccurately duplicated in the descriptions of functions and structures using the **sigevent** structure. The revised wording does not relax the requirement that the signal number be in the range SIGRTMIN to SIGRTMAX to guarantee queuing and passing of the application value, since that requirement is still implied by the signal names.

- IEEE Std. 1003.1-200x is intentionally vague on whether ''non-realtime'' signal-generating mechanisms can result in a **siginfo_t** being supplied to the handler on delivery. In one existing implementation, a **siginfo_t** is posted on signal generation, even though the implementation does not support queuing of multiple occurrences of a signal. It is not the intent of IEEE Std. 1003.1-200x to preclude this, independent of the mandate to define signals that do support queuing. Any interpretation that appears to preclude this is a mistake in the reading or writing of the standard.

- Signals handled by realtime signal handlers might be generated by functions or conditions that do not allow the specification of an application-defined value and do not queue. IEEE Std. 1003.1-200x specifies the *si_code* member of the **siginfo_t** structure used in existing practice and defines additional codes so that applications can detect whether an application-defined value is present or not. The code SI_USER for *kill*( )-generated signals is adopted from existing practice.

- The *sigaction*( ) *sa_flags* value SA_SIGINFO tells the implementation that the signal-catching function expects two additional arguments. When the flag is not set, a single argument, the signal number, is passed as specified by IEEE Std. 1003.1-200x. Although IEEE Std. 1003.1-200x does not explicitly allow the *info* argument to the handler function to be NULL, this is existing practice. This provides for compatibility with programs whose signal-catching functions are not prepared to accept the additional arguments. IEEE Std. 1003.1-200x is explicitly unspecified as to whether signals actually queue when SA_SIGINFO is not set for a signal, as there appear to be no benefits to applications in specifying one behavior or another. One existing implementation queues a **siginfo_t** on each signal generation, unless the signal is already pending, in which case the implementation discards the new **siginfo_t**; that is, the queue length is never greater than one. This implementation only examines SA_SIGINFO on signal delivery, discarding the queued **siginfo_t** if its delivery was not requested.

  IEEE Std. 1003.1-200x specifies several new values for the *si_code* member of the **siginfo_t** structure. In existing practice, a *si_code* value of less than or equal to zero indicates that the signal was generated by a process via the *kill*( ) function. In existing practice, values of *si_code* that provide additional information for implementation-generated signals, such as SIGFPE or SIGSEGV , are all positive. Thus, if implementations define the new constants specified in IEEE Std. 1003.1-200x to be negative numbers, programs written to use existing practice will not break. IEEE Std. 1003.1-200x chose not to attempt to specify existing practice values of *si_code* other than SI_USER both because it was deemed beyond the scope of IEEE Std. 1003.1-200x and because many of the values in existing practice appear to be platform and implementation-defined. But, IEEE Std. 1003.1-200x does specify that if an implementation—for example, one that does not have existing practice in this area—chooses to define additional values for *si_code*, these values have to be different from the values of the symbols specified by IEEE Std. 1003.1-200x. This will allow portable applications to differentiate between signals generated by one of the POSIX.1b asynchronous events and those generated by other implementation events in a manner compatible with existing practice.

  The unique values of *si_code* for the POSIX.1b asynchronous events have implications for implementations of, for example, asynchronous I/O or message passing in user space library code. Such an implementation will be required to provide a hidden interface to the signal generation mechanism that allows the library to specify the standard values of *si_code*.

  Existing practice also defines additional members of **siginfo_t**, such as the process ID and user ID of the sending process for *kill*( )-generated signals. These members were deemed not necessary to meet the requirements of realtime applications and are not specified by IEEE Std. 1003.1-200x. Neither are they precluded.

  The third argument to the signal-catching function, *context*, is left undefined by IEEE Std. 1003.1-200x, but is specified in the interface because it matches existing practice for the SA_SIGINFO flag. It was considered undesirable to require a separate implementation for SA_SIGINFO for POSIX conformance on implementations that already support the two additional parameters.

- The requirement to deliver lower numbered signals in the range SIGRTMIN to SIGRTMAX first, when multiple unblocked signals are pending, results from several considerations:

  — A method is required to prioritize event notifications. The signal number was chosen instead of, for instance, associating a separate priority with each request, because an implementation has to check pending signals at various points and select one for delivery when more than one is pending. Specifying a selection order is the minimal additional semantic that will achieve prioritized delivery. If a separate priority were to be associated with queued signals, it would be necessary for an implementation to search all non-empty, non-blocked signal queues and select from among them the pending signal with the highest priority. This would significantly increase the cost of and decrease the determinism of signal delivery.

— Given the specified selection of the lowest numeric unblocked pending signal, preemptive priority signal delivery can be achieved using signal numbers and signal masks by ensuring that the *sa_mask* for each signal number blocks all signals with a higher numeric value.

For realtime applications that want to use only the newly defined realtime signal numbers without interference from the standard signals, this can be achieved by blocking all of the standard signals in the process signal mask and in the *sa_mask* installed by the signal action for the realtime signal handlers.

IEEE Std. 1003.1-200x explicitly leaves unspecified the ordering of signals outside of the range of realtime signals and the ordering of signals within this range with respect to those outside the range. It was believed that this would unduly constrain implementations or standards in the future definition of new signals.

B.2.4.3 Signal Actions

Early proposals mentioned SIGCONT as a second exception to the rule that signals are not delivered to stopped processes until continued. Because IEEE Std. 1003.1-200x now specifies that SIGCONT causes the stopped process to continue when it is generated, delivery of SIGCONT is not prevented because a process is stopped, even without an explicit exception to this rule.

Ignoring a signal by setting the action to SIG_IGN (or SIG_DFL for signals whose default action is to ignore) is not the same as installing a signal-catching function that simply returns. Invoking such a function will interrupt certain system functions that block processes (for example, *wait*( ), *sigsuspend*( ), *pause*( ), *read*( ), *write*( )) while ignoring a signal has no such effect on the process.

Historical implementations discard pending signals when the action is set to SIG_IGN. However, they do not always do the same when the action is set to SIG_DFL and the default action is to ignore the signal. IEEE Std. 1003.1-200x requires this for the sake of consistency and also for completeness, since the only signal this applies to is SIGCHLD, and IEEE Std. 1003.1-200x disallows setting its action to SIG_IGN.

The specification of the effects of SIG_IGN on SIGCHLD as implementation-defined permits, but does not require, the System V effect of causing terminating children to be ignored by *wait*( ). Yet it permits SIGCHLD to be effectively ignored in an implementation-defined manner by use of SIG_DFL.

Some implementations (System V, for example) assign different semantics for SIGCLD depending on whether the action is set to SIG_IGN or SIG_DFL. Since POSIX.1 requires that the default action for SIGCHLD be to ignore the signal, applications should always set the action to SIG_DFL in order to avoid SIGCHLD.

Some implementations (System V, for example) will deliver a SIGCLD signal immediately when a process establishes a signal-catching function for SIGCLD when that process has a child that has already terminated. Other implementations, such as 4.3 BSD, do not generate a new SIGCHLD signal in this way. In general, a process should not attempt to alter the signal action for the SIGCHLD signal while it has any outstanding children. However, it is not always possible for a process to avoid this; for example, shells sometimes start up processes in pipelines with other processes from the pipeline as children. Processes that cannot ensure that they have no children when altering the signal action for SIGCHLD thus need to be prepared for, but not depend on, generation of an immediate SIGCHLD signal.

The default action of the stop signals (SIGSTOP , SIGTSTP, SIGTTIN, SIGTTOU) is to stop a process that is executing. If a stop signal is delivered to a process that is already stopped, it has no effect. In fact, if a stop signal is generated for a stopped process whose signal mask blocks the signal, the signal will never be delivered to the process since the process must receive a SIGCONT, which discards all pending stop signals, in order to continue executing.

The SIGCONT signal shall continue a stopped process even if SIGCONT is blocked (or ignored). However, if a signal-catching routine has been established for SIGCONT, it will not be entered until SIGCONT is unblocked.

If a process in an orphaned process group stops, it is no longer under the control of a job control shell and hence would not normally ever be continued. Because of this, orphaned processes that receive terminal-related stop signals (SIGTSTP , SIGTTIN, SIGTTOU, but not SIGSTOP) must not be allowed to stop. The goal is to prevent stopped processes from languishing forever. (As SIGSTOP is sent only via *kill*( ), it is assumed that the process or user sending a SIGSTOP can send a SIGCONT when desired.) Instead, the system must discard the stop signal. As an extension, it may also deliver another signal in its place. 4.3 BSD sends a SIGKILL, which is overly effective because SIGKILL is not catchable. Another possible choice is SIGHUP. 4.3 BSD also does this for orphaned processes (processes whose parent has terminated) rather than for members of orphaned process groups; this is less desirable because job control shells manage process groups. POSIX.1 also prevents SIGTTIN and SIGTTOU signals from being generated for processes in orphaned process groups as a direct result of activity on a terminal, preventing infinite loops when *read*( ) and *write*( ) calls generate signals that are discarded; see Section A.11.1.4 (on page 3371). A similar restriction on the generation of SIGTSTP was considered, but that would be unnecessary and more difficult to implement due to its asynchronous nature.

Although POSIX.1 requires that signal-catching functions be called with only one argument, there is nothing to prevent conforming implementations from extending POSIX.1 to pass additional arguments, as long as Strictly Conforming POSIX.1 Applications continue to compile and execute correctly. Most historical implementations do, in fact, pass additional, signal-specific arguments to certain signal-catching routines.

There was a proposal to change the declared type of the signal handler to:

```
void func (int sig, ...);
```

The usage of ellipses (`"..."`) is ISO C standard syntax to indicate a variable number of arguments. Its use was intended to allow the implementation to pass additional information to the signal handler in a standard manner.

Unfortunately, this construct would require all signal handlers to be defined with this syntax because the ISO C standard allows implementations to use a different parameter passing mechanism for variable parameter lists than for non-variable parameter lists. Thus, all existing signal handlers in all existing applications would have to be changed to use the variable syntax in order to be standard and portable. This is in conflict with the goal of Minimal Changes to Existing Application Code.

When terminating a process from a signal-catching function, processes should be aware of any interpretation that their parent may make of the status returned by *wait*( ) or *waitpid*( ). In particular, a signal-catching function should not call *exit*(0) or *_exit*(0) unless it wants to indicate successful termination. A non-zero argument to *exit*( ) or *_exit*( ) can be used to indicate unsuccessful termination. Alternatively, the process can use *kill*( ) to send itself a fatal signal (first ensuring that the signal is set to the default action and not blocked). See also the

RATIONALE section of the *_exit*( ) function.

The behavior of *unsafe* functions, as defined by this section, is undefined when they are invoked from signal-catching functions in certain circumstances. The behavior of reentrant functions, as defined by this section, is as specified by POSIX.1, 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 restriction. 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*( ) (see *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. POSIX.1 does not address the more general problem of synchronizing access to shared data structures. Note in particular that even the ''safe'' functions may modify the global variable *errno*; the signal-catching function may want to save and restore its value. 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*( ) or *siglongjmp*( ) can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use *longjmp*( ) or *siglongjmp*( ) out of 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 the C language *malloc*( ) or *free*( ) functions or the ISO C standard I/O library, both of which traditionally use data structures in a non-reentrant manner. Because any combination of different functions using a common data structure can cause reentrancy problems, POSIX.1 does not define the behavior when any unsafe function is called in a signal handler that interrupts any unsafe function.

The only realtime extension to signal actions is the addition of the additional parameters to the signal-catching function. This extension has been explained and motivated in the previous section. In making this extension, though, developers of POSIX.1b ran into issues relating to function prototypes. In response to input from the POSIX.1 standard developers, members were added to the **sigaction** structure to specify function prototypes for the newer signal-catching function specified by POSIX.1b. These members follow changes that are being made to POSIX.1. Note that IEEE Std. 1003.1-200x explicitly states that these fields may overlap so that a union can be defined. This will enable existing implementations of POSIX.1 to maintain binary-compatibility when these extensions are added.

The **siginfo_t** structure was adopted for passing the application-defined value to match existing practice, but the existing practice has no provision for an application-defined value, so this was added. Note that POSIX normally reserves the ''_t'' type designation for opaque types. The **siginfo_t** structure breaks with this convention to follow existing practice and thus promote portability. Standardization of the existing practice for the other members of this structure may be addressed in the future.

Although it is not explicitly visible to applications, there are additional semantics for signal actions implied by queued signals and their interaction with other POSIX.1b realtime functions. Specifically:

- It is not necessary to queue signals whose action is SIG_IGN.

- For implementations that support POSIX.1b timers, some interaction with the timer functions at signal delivery is implied to manage the timer overrun count.

B.2.4.4 Signal Effects on Other Functions

The most common behavior of an interrupted function after a signal-catching function returns is for the interrupted function to give an [EINTR] error. However, there are a number of specific exceptions, including *sleep*( ) and certain situations with *read*( ) and *write*( ).

The historical implementations of many functions defined by IEEE Std. 1003.1-200x are not interruptible, but delay delivery of signals generated during their execution until after they complete. This is never a problem for functions that are guaranteed to complete in a short (imperceptible to a human) period of time. It is normally those functions that can suspend a process indefinitely or for long periods of time (for example, *wait*( ), *pause*( ), *sigsuspend*( ), *sleep*( ), or *read*( )/*write*( ) on a slow device like a terminal] that are interruptible. This permits applications to respond to interactive signals or to set timeouts on calls to most such functions with *alarm*( ). Therefore, implementations should generally make such functions (including ones defined as extensions) interruptible.

Functions not mentioned explicitly as interruptible may be so on some implementations, possibly as an extension where the function gives an [EINTR] error. There are several functions (for example, *getpid*( ), *getuid*( )) that are specified as never returning an error, which can thus never be extended in this way.

B.2.5 Standard I/O Streams

B.2.5.1 Interaction of File Descriptors and Standard I/O Streams

There is no additional rationale for this section.

B.2.5.2 Stream Orientation and Encoding Rules

There is no additional rationale for this section.

B.2.6 STREAMS

STREAMS are introduced into IEEE Std. 1003.1-200x as part of the alignment with the Single UNIX Specification, but marked as an option in recognition that not all systems may wish to implement the facility. The option within IEEE Std. 1003.1-200x is denoted by the XSR margin marker. The standard developers made this option independent of the XSI option.

STREAMS are a method of implementing network services and other character-based input/output mechanisms, with the STREAM being a full-duplex connection between a process and a device. STREAMS provides direct access to protocol modules, and optional protocol modules can be interposed between the process-end of the STREAM and the device-driver at the device-end of the STREAM. Pipes can be implemented using the STREAMS mechanism, so they can provide process-to-process as well as process-to-device communications.

This section introduces STREAMS I/O, the message types used to control them, an overview of the priority mechanism, and the interfaces used to access them.

B.2.6.1 *Accessing STREAMS*

There is no additional rationale for this section.

B.2.7 XSI Interprocess Communication

There are two forms of IPC supported as options in IEEE Std. 1003.1-200x. The traditional System V IPC routines derived from the SVID—that is, the *msg**( ), *sem**( ), and *shm**( ) interfaces—are mandatory on XSI-conformant systems. Thus, all XSI-conformant systems provide the same mechanisms for manipulating messages, shared memory, and semaphores.

In addition, the POSIX Realtime Extension provides an alternate set of routines for those systems supporting the appropriate options.

For maximum portability to UNIX systems, the former are recommended. However, if the target for an application is a realtime system, then application developers are advised to write their code in such a way that modules using IPC interfaces can be modified easily in the future to use either interfaces.

B.2.7.1 *IPC General Information*

General information that is shared by all three mechanisms is described in this section. The common permissions mechanism is briefly introduced, describing the mode bits, and how they are used to determine whether or not a process has access to read or write/alter the appropriate instance of one of the IPC mechanisms. All other relevant information is contained in the reference pages themselves.

The semaphore type of IPC allows processes to communicate through the exchange of semaphore values. A semaphore is a positive integer. Since many applications require the use of more than one semaphore, XSI-conformant systems have the ability to create sets or arrays of semaphores.

Calls to support semaphores include:

semctl( ), *semget*( ), *semop*( )

Semaphore sets are created by using the *semget*( ) function.

The message type of IPC allows process to communicate through the exchange of data stored in buffers. This data is transmitted between processes in discrete portions known as messages.

Calls to support message queues include:

msgctl( ), *msgget*( ), *msgrcv*( ), *msgsnd*( )

The share memory type of IPC allows two or more processes to share memory and consequently the data contained therein. This is done by allowing processes to set up access to a common memory address space. This sharing of memory provides a fast means of exchange of data between processes.

Calls to support shared memory include:

shmctl( ), *shmdt*( ), *shmget*( )

The *ftok*( ) interface is also provided.

B.2.8 Realtime

Advisory Information

POSIX.1b contains an Informative Annex with proposed interfaces for ''real-time files''. These interfaces could determine groups of the exact parameters required to do ''direct I/O'' or ''extents''. These interfaces were objected to by a significant portion of the balloting group as too complex. A portable application had little chance of correctly navigating the large parameter space to match its desires to the system. In addition, they only applied to a new type of file (realtime files) and they told the implementation exactly what to do as opposed to advising the implementation on application behavior and letting it optimize for the system the (portable) application was running on. For example, it was not clear how a system that had a disk array should set its parameters.

There seemed to be several overall goals:

- Optimizing sequential access
- Optimizing caching behavior
- Optimizing I/O data transfer
- Preallocation

The advisory interfaces, *posix_fadvise*( ) and *posix_madvise*( ), satisfy the first two goals. The POSIX_FADV_SEQUENTIAL and POSIX_MADV_SEQUENTIAL advice tells the implementation to expect serial access. Typically the system will prefetch the next several serial accesses in order to overlap I/O. It may also free previously accessed serial data if memory is tight. If the application is not doing serial access it can use POSIX_FADV_WILLNEED and POSIX_MADV_WILLNEED to accomplish I/O overlap, as required. When the application advises POSIX_FADV_RANDOM or POSIX_MADV_RANDOM behavior, the implementation usually tries to fetch a minimum amount of data with each request and it does not expect much locality. POSIX_FADV_DONTNEED and POSIX_MADV_DONTNEED allow the system to free up caching resources as the data will not be required in the near future.

POSIX_FADV_NOREUSE tells the system that caching the specified data is not optimal. For file I/O, the transfer should go directly to the user buffer instead of being cached internally by the implementation. To portably perform direct disk I/O on all systems, the application must perform its I/O transfers according to the following rules:

1. The user buffer should be aligned according to the {POSIX_REC_XFER_ALIGN} *pathconf*( ) variable.
2. The number of bytes transferred in an I/O operation should be a multiple of the {POSIX_ALLOC_SIZE_MIN} *pathconf*( ) variable.
3. The offset into the file at the start of an I/O operation should be a multiple of the {POSIX_ALLOC_SIZE_MIN} *pathconf*( ) variable.
4. The application should ensure that all threads which open a given file specify POSIX_FADV_NOREUSE to be sure that there is no unexpected interaction between threads using buffered I/O and threads using direct I/O to the same file.

In some cases, a user buffer must be properly aligned in order to be transferred directly to/from the device. The {POSIX_REC_XFER_ALIGN} *pathconf*( ) variable tells the application the proper alignment.

The preallocation goal is met by the space control function, *posix_fallocate*( ). The application can use *posix_fallocate*( ) to guarantee no [ENOSPC] errors and to improve performance by prepaying any overhead required for block allocation.

Implementations may use information conveyed by a previous *posix_fadvise*( ) call to influence the manner in which allocation is performed. For example, if an application did the following calls:

```
fd = open("file");
posix_fadvise(fd, offset, len, POSIX_FADV_SEQUENTIAL);
posix_fallocate(fd, len, size);
```

an implementation might allocate the file contiguously on disk.

Finally, the *pathconf*( ) variables {POSIX_REC_MIN_XFER_SIZE}, {POSIX_REC_MAX_XFER_SIZE}, and {POSIX_REC_INCR_XFER_SIZE} tell the application a range of transfer sizes that are recommended for best I/O performance.

Where bounded response time is required, the vendor can supply the appropriate settings of the advisories to achieve a guaranteed performance level.

The interfaces meet the goals while allowing applications using regular files to take advantage of performance optimizations. The interfaces tell the implementation expected application behavior which the implementation can use to optimize performance on a particular system with a particular dynamic load.

The *posix_memalign*( ) function was added to allow for the allocation of specifically aligned buffers; for example, for {POSIX_REC_XFER_ALIGN}.

The working group also considered the alternative of adding a function which would return an aligned pointer to memory within a user supplied buffer. This was not considered to be the best method, because it potentially wastes large amounts of memory when buffers need to be aligned on large alignment boundaries.

Message Passing

This section provides the rationale for the definition of the message passing interface in IEEE Std. 1003.1-200x. This is presented in terms of the objectives, models, and requirements imposed upon this interface.

- Objectives

  Many applications, including both realtime and database applications, require a means of passing arbitrary amounts of data between cooperating processes comprising the overall application on one or more processors. Many conventional interfaces for interprocess communication are insufficient for realtime applications in that efficient and deterministic data passing methods cannot be implemented. This has prompted the definition of message passing interfaces providing these facilities:

  — Open a message queue.

  — Send a message to a message queue.

  — Receive a message from a queue, either synchronously or asynchronously.

  — Alter message queue attributes for flow and resource control.

  It is assumed that an application may consist of multiple cooperating processes and that these processes may wish to communicate and coordinate their activities. The message passing facility described in IEEE Std. 1003.1-200x allows processes to communicate through system-wide queues. These message queues are accessed through names that may be path names. A message queue can be opened for use by multiple sending and/or multiple receiving processes.

- Background on Embedded Applications

  Interprocess communication utilizing message passing is a key facility for the construction of deterministic, high-performance realtime applications. The facility is present in all realtime systems and is the framework upon which the application is constructed. The performance of the facility is usually a direct indication of the performance of the resulting application.

  Realtime applications, especially for embedded systems, are typically designed around the performance constraints imposed by the message passing mechanisms. Applications for embedded systems are typically very tightly constrained. Application writers expect to design and control the entire system. In order to minimize system costs, the writer will attempt to use all resources to their utmost and minimize the requirement to add additional memory or processors.

  The embedded applications usually share address spaces and only a simple message passing mechanism is required. The application can readily access common data incurring only mutual-exclusion overheads. The models desired are the simplest possible with the application building higher-level facilities only when needed.

- Requirements

  The following requirements determined the features of the message passing facilities defined in IEEE Std. 1003.1-200x:

  — Naming of Message Queues

    The mechanism for gaining access to a message queue is a path name evaluated in a context that is allowed to be a file system name space, or it can be independent of any file system. This is a specific attempt to allow implementations based on either method in order to address both embedded systems and to also allow implementation in larger systems.

    The interface of *mq_open*( ) is defined to allow but not require the access control and name conflicts resulting from utilizing a file system for name resolution. All required behavior is specified for the access control case. Yet a conforming implementation, such as an embedded system kernel, may define that there are no distinctions between users and may define that all process have all access privileges.

  — Embedded System Naming

    Embedded systems need to be able to utilize independent name spaces for accessing the various system objects. They typically do not have a file system, precluding its utilization as a common name resolution mechanism. The modularity of an embedded system limits the connections between separate mechanisms that can be allowed.

    Embedded systems typically do not have any access protection. Since the system does not support the mixing of applications from different areas, and usually does not even have the concept of an authorization entity, access control is not useful.

  — Large System Naming

    On systems with more functionality, the name resolution must support the ability to use the file system as the name resolution mechanism/object storage medium and to have control over access to the objects. Utilizing the path name space can result in further errors when the names conflict with other objects.

  — Fixed Size of Messages

The interfaces impose a fixed upper bound on the size of messages that can be sent to a specific message queue. The size is set on an individual queue basis and cannot be changed dynamically.

The purpose of the fixed size is to increase the ability of the system to optimize the implementation of *mq_send*( ) and *mq_receive*( ). With fixed sizes of messages and fixed numbers of messages, specific message blocks can be pre-allocated. This eliminates a significant amount of checking for errors and boundary conditions. Additionally, an implementation can optimize data copying to maximize performance. Finally, with a restricted range of message sizes, an implementation is better able to provide deterministic operations.

— Prioritization of Messages

Message prioritization allows the application to determine the order in which messages are received. Prioritization of messages is a key facility that is provided by most realtime kernels and is heavily utilized by the applications. The major purpose of having priorities in message queues is to avoid priority inversions in the message system, where a high-priority message is delayed behind one or more lower-priority messages. It has been observed that a significant problem with Ada rendezvous is that it queues tasks in strict FIFO order, ignoring priorities. This allows the applications to be designed so that they do not need to be interrupted in order to change the flow of control when exceptional conditions occur. The prioritization does add additional overhead to the message operations in those cases it is actually used but a clever implementation can optimize for the FIFO case to make that more efficient.

— Asynchronous Notification

The interface supports the ability to have a task asynchronously notified of the availability of a message on the queue. The purpose of this facility is to allow the task to perform other functions and yet still be notified that a message has become available on the queue.

To understand the requirement for this function, it is useful to understand two models of application design: a single task performing multiple functions and multiple tasks performing a single function. Each of these models has advantages.

Asynchronous notification is required to build the model of a single task performing multiple operations. This model typically results from either the expectation that interruption is less expensive than utilizing a separate task or from the growth of the application to include additional functions.

Semaphores

Semaphores are a high-performance process synchronization mechanism. Semaphores are named by null-terminated strings of characters.

A semaphore is created using the *sem_init*( ) function or the *sem_open*( ) function with the O_CREAT flag set in *oflag*.

To use a semaphore, a process has to first initialize the semaphore or inherit an open descriptor for the semaphore via *fork*( ).

A semaphore preserves its state when the last reference is closed. For example, if a semaphore has a value of 13 when the last reference is closed, it will have a value of 13 when it is next opened.

When a semaphore is created, an initial state for the semaphore has to be provided. This value is a non-negative integer. Negative values are not possible since they indicate the presence of blocked processes. The persistence of any of these objects across a system crash or a system reboot is undefined. Conforming applications shall not depend on any sort of persistence across a system reboot or a system crash.

- Models and Requirements

  A realtime system requires synchronization and communication between the processes comprising the overall application. An efficient and reliable synchronization mechanism has to be provided in a realtime system that will allow more than one schedulable process mutually-exclusive access to the same resource. This synchronization mechanism has to allow for the optimal implementation of synchronization or systems implementors will define other, more cost-effective methods.

  At issue are the methods whereby multiple processes (tasks) can be designed and implemented to work together in order to perform a single function. This requires interprocess communication and synchronization. A semaphore mechanism is the lowest level of synchronization that can be provided by an operating system.

  A semaphore is defined as an object that has an integral value and a set of blocked processes associated with it. If the value is positive or zero, then the set of blocked processes is empty; otherwise, the size of the set is equal to the absolute value of the semaphore value. The value of the semaphore can be incremented or decremented by any process with access to the semaphore and must be done as an indivisible operation. When a semaphore value is less than or equal to zero, any process that attempts to lock it again will block or be informed that it is not possible to perform the operation.

  A semaphore may be used to guard access to any resource accessible by more than one schedulable task in the system. It is a global entity and not associated with any particular process. As such, a method of obtaining access to the semaphore has to be provided by the operating system. A process that wants access to a critical resource (section) has to wait on the semaphore that guards that resource. When the semaphore is locked on behalf of a process, it knows that it can utilize the resource without interference by any other cooperating process in the system. When the process finishes its operation on the resource, leaving it in a well-defined state, it posts the semaphore, indicating that some other process may now obtain the resource associated with that semaphore.

  In this section, mutexes and condition variables are specified as the synchronization mechanisms between threads.

  These primitives are typically used for synchronizing threads that share memory in a single process. However, this section provides an option allowing the use of these synchronization interfaces and objects between processes that share memory, regardless of the method for sharing memory.

  Much experience with semaphores shows that there are two distinct uses of synchronization: locking, which is typically of short duration; and waiting, which is typically of long or unbounded duration. These distinct usages map directly onto mutexes and condition variables, respectively.

  Semaphores are provided in IEEE Std. 1003.1-200x primarily to provide a means of synchronization for processes; these processes may or may not share memory. Mutexes and condition variables are specified as synchronization mechanisms between threads; these threads always share (some) memory. Both are synchronization paradigms that have been in widespread use for a number of years. Each set of primitives is particularly well matched to certain problems.

With respect to binary semaphores, experience has shown that condition variables and mutexes are easier to use for many synchronization problems than binary semaphores. The primary reason for this is the explicit appearance of a Boolean predicate that specifies when the condition wait is satisfied. This Boolean predicate terminates a loop, including the call to *pthread_cond_wait*( ). As a result, extra wakeups are benign since the predicate governs whether the thread will actually proceed past the condition wait. With stateful primitives, such as binary semaphores, the wakeup in itself typically means that the wait is satisfied. The burden of ensuring correctness for such waits is thus placed on *all* signalers of the semaphore rather than on an *explicitly coded* Boolean predicate located at the condition wait. Experience has shown that the latter creates a major improvement in safety and ease-of-use.

Counting semaphores are well matched to dealing with producer/consumer problems, including those that might exist between threads of different processes, or between a signal handler and a thread. In the former case, there may be little or no memory shared by the processes; in the latter case, one is not communicating between co-equal threads, but between a thread and an interruptlike entity. It is for these reasons that IEEE Std. 1003.1-200x allows semaphores to be used by threads.

Mutexes and condition variables have been effectively used with and without priority inheritance, priority ceiling, and other attributes to synchronize threads that share memory. The efficiency of their implementation is comparable to or better than that of other synchronization primitives that are sometimes harder to use (for example, binary semaphores). Furthermore, there is at least one known implementation of Ada tasking that uses these primitives. Mutexes and condition variables together constitute an appropriate, sufficient, and complete set of interthread synchronization primitives.

Efficient multi-threaded applications require high-performance synchronization primitives. Considerations of efficiency and generality require a small set of primitives upon which more sophisticated synchronization functions can be built.

- Standardization Issues

  It is possible to implement very high-performance semaphores using test-and-set instructions on shared memory locations. The library routines that implement such a high-performance interface has to properly ensure that a *sem_wait*( ) or *sem_trywait*( ) operation that cannot be performed will issue a blocking semaphore system call or properly report the condition to the application. The same interface to the application program would be provided by a high-performance implementation.

B.2.8.1 Realtime Signals

Realtime Signals Extension

This portion of the rationale presents models, requirements, and standardization issues relevant to the Realtime Signals Extension. This extension provides the capability required to support reliable, deterministic, asynchronous notification of events. While a new mechanism, unencumbered by the historical usage and semantics of POSIX.1 signals, might allow for a more efficient implementation, the application requirements for event notification can be met with a small number of extensions to signals. Therefore, a minimal set of extensions to signals to support the application requirements is specified.

The realtime signal extensions specified in this section are used by other realtime functions requiring asynchronous notification:

- Models

The model supported is one of multiple cooperating processes, each of which handles multiple asynchronous external events. Events represent occurrences that are generated as the result of some activity in the system. Examples of occurrences that can constitute an event include:

— Completion of an asynchronous I/O request

— Expiration of a POSIX.1b timer

— Arrival of an interprocess message

— Generation of a user-defined event

Processing of these events may occur synchronously via polling for event notifications or asynchronously via a software interrupt mechanism. Existing practice for this model is well established for traditional proprietary realtime operating systems, realtime executives, and realtime extended POSIX-like systems.

A contrasting model is that of ''cooperating sequential processes'' where each process handles a single priority of events via polling. Each process blocks while waiting for events, and each process depends on the preemptive, priority-based process scheduling mechanism to arbitrate between events of different priority that need to be processed concurrently. Existing practice for this model is also well established for small realtime executives that typically execute in an unprotected physical address space, but it is just emerging in the context of a fuller function operating system with multiple virtual address spaces.

It could be argued that the cooperating sequential process model, and the facilities supported by the POSIX Threads Extension obviate a software interrupt model. But, even with the cooperating sequential process model, the need has been recognized for a software interrupt model to handle exceptional conditions and process aborting, so the mechanism must be supported in any case. Furthermore, it is not the purview of IEEE Std. 1003.1-200x to attempt to convince realtime practitioners that their current application models based on software interrupts are ''broken'' and should be replaced by the cooperating sequential process model. Rather, it is the charter of IEEE Std. 1003.1-200x to provide standard extensions to mechanisms that support existing realtime practice.

- Requirements

This section discusses the following realtime application requirements for asynchronous event notification:

— Reliable delivery of asynchronous event notification

The events notification mechanism shall guarantee delivery of an event notification. Asynchronous operations (such as asynchronous I/O and timers) that complete significantly after they are invoked have to guarantee that delivery of the event notification can occur at the time of completion.

— Prioritized handling of asynchronous event notifications

The events notification mechanism shall support the assigning of a user function as an event notification handler. Furthermore, the mechanism shall support the preemption of an event handler function by a higher priority event notification and shall support the selection of the highest priority pending event notification when multiple notifications (of different priority) are pending simultaneously.

The model here is based on hardware interrupts. Asynchronous event handling allows the application to ensure that time-critical events are immediately processed when delivered, without the indeterminism of being at a random location within a polling loop.

Use of handler priority allows the specification of how handlers are interrupted by other higher priority handlers.

— Differentiation between multiple occurrences of event notifications of the same type

The events notification mechanism shall pass an application-defined value to the event handler function. This value can be used for a variety of purposes, such as enabling the application to identify which of several possible events of the same type (for example, timer expirations) has occurred.

— Polled reception of asynchronous event notifications

The events notification mechanism shall support blocking and non-blocking polls for asynchronous event notification.

The polled mode of operation is often preferred over the interrupt mode by those practitioners accustomed to this model. Providing support for this model facilitates the porting of applications based on this model to POSIX.1b conforming systems.

— Deterministic response to asynchronous event notifications

The events notification mechanism shall not preclude implementations that provide deterministic event dispatch latency and shall minimize the number of system calls needed to use the event facilities during realtime processing.

- Rationale for Extension

  POSIX.1 signals have many of the characteristics necessary to support the asynchronous handling of event notifications, and the Realtime Signals Extension addresses the following deficiencies in the POSIX.1 signal mechanism:

  — Signals do not support reliable delivery of event notification. Subsequent occurrences of a pending signal are not guaranteed to be delivered.

  — Signals do not support prioritized delivery of event notifications. The order of signal delivery when multiple unblocked signals are pending is undefined.

  — Signals do not support the differentiation between multiple signals of the same type.

B.2.8.2 Asynchronous I/O

Many applications need to interact with the I/O subsystem in an asynchronous manner. The asynchronous I/O mechanism provides the ability to overlap application processing and I/O operations initiated by the application. The asynchronous I/O mechanism allows a single process to perform I/O simultaneously to a single file multiple times or to multiple files multiple times.

Overview

Asynchronous I/O operations proceed in logical parallel with the processing done by the application after the asynchronous I/O has been initiated. Other than this difference, asynchronous I/O behaves similarly to normal I/O using *read*( ), *write*( ), *lseek*( ), and *fsync*( ). The effect of issuing an asynchronous I/O request is as if a separate thread of execution were to perform atomically the implied *lseek*( ) operation, if any, and then the requested I/O operation (either *read*( ), *write*( ), or *fsync*( )). There is no seek implied with a call to *aio_fsync*( ). Concurrent asynchronous operations and synchronous operations applied to the same file update the file as if the I/O operations had proceeded serially.

When asynchronous I/O completes, a signal can be delivered to the application to indicate the completion of the I/O. This signal can be used to indicate that buffers and control blocks used for asynchronous I/O can be reused. Signal delivery is not required for an asynchronous operation and may be turned off on a per-operation basis by the application. Signals may also be synchronously polled using *aio_suspend*( ), *sigtimedwait*( ), or *sigwaitinfo*( ).

Normal I/O has a return value and an error status associated with it. Asynchronous I/O returns a value and an error status when the operation is first submitted, but that only relates to whether the operation was successfully queued up for servicing. The I/O operation itself also has a return status and an error value. To allow the application to retrieve the return status and the error value, functions are provided that, given the address of an asynchronous I/O control block, yield the return and error status associated with the operation. Until an asynchronous I/O operation is done, its error status shall be [EINPROGRESS]. Thus, an application can poll for completion of an asynchronous I/O operation by waiting for the error status to become equal to a value other than [EINPROGRESS]. The return status of an asynchronous I/O operation is undefined so long as the error status is equal to [EINPROGRESS].

Storage for asynchronous operation return and error status may be limited. Submission of asynchronous I/O operations may fail if this storage is exceeded. When an application retrieves the return status of a given asynchronous operation, therefore, any system-maintained storage used for this status and the error status may be reclaimed for use by other asynchronous operations.

Asynchronous I/O can be performed on file descriptors that have been enabled for POSIX.1b synchronized I/O. In this case, the I/O operation still occurs asynchronously, as defined herein; however, the asynchronous operation I/O in this case is not completed until the I/O has reached either the state of synchronized I/O data integrity completion or synchronized I/O file integrity completion, depending on the sort of synchronized I/O that is enabled on the file descriptor.

Models

Three models illustrate the use of asynchronous I/O: a journalization model, a data acquisition model, and a model of the use of asynchronous I/O in supercomputing applications.

- Journalization Model

  Many realtime applications perform low-priority journalizing functions. Journalizing requires that logging records be queued for output without blocking the initiating process.

- Data Acquisition Model

  A data acquisition process may also serve as a model. The process has two or more channels delivering intermittent data that must be read within a certain time. The process issues one asynchronous read on each channel. When one of the channels needs data collection, the process reads the data and posts it through an asynchronous write to secondary memory for future processing.

- Supercomputing Model

  The supercomputing community has used asynchronous I/O much like that specified herein for many years. This community requires the ability to perform multiple I/O operations to multiple devices with a minimal number of entries to ‘‘the system’’; each entry to ‘‘the system’’ provokes a major delay in operations when compared to the normal progress made by the application. This existing practice motivated the use of combined *lseek*( ) and *read*( ) or *write*( ) calls, as well as the *lio_listio*( ) call. Another common practice is to disable signal notification for I/O completion, and simply poll for I/O completion at some interval by which the I/O should be completed. Likewise, interfaces like *aio_cancel*( ) have been in successful commercial use for many years. Note also that an underlying implementation of asynchronous I/O will require the ability, at least internally, to cancel outstanding asynchronous I/O, at least when the process exits. (Consider an asynchronous read from a terminal, when the process intends to exit immediately.)

Requirements

Asynchronous input and output for realtime implementations have these requirements:

- The ability to queue multiple asynchronous read and write operations to a single open instance. Both sequential and random access should be supported.
- The ability to queue asynchronous read and write operations to multiple open instances.
- The ability to obtain completion status information by polling and/or asynchronous event notification.
- Asynchronous event notification on asynchronous I/O completion is optional.
- It has to be possible for the application to associate the event with the *aiocbp* for the operation that generated the event.
- The ability to cancel queued requests.
- The ability to wait upon asynchronous I/O completion in conjunction with other types of events.
- The ability to accept an *aio_read*( ) and an *aio_cancel*( ) for a device that accepts a *read*( ), and the ability to accept an *aio_write*( ) and an *aio_cancel*( ) for a device that accepts a *write*( ). This does not imply that the operation is asynchronous.

Standardization Issues

The following issues are addressed by the standardization of asynchronous I/O:

- Rationale for New Interface

  Non-blocking I/O does not satisfy the needs of either realtime or high-performance computing models; these models require that a process overlap program execution and I/O processing. Realtime applications will often make use of direct I/O to or from the address space of the process, or require synchronized (unbuffered) I/O; they also require the ability to overlap this I/O with other computation. In addition, asynchronous I/O allows an application to keep a device busy at all times, possibly achieving greater throughput. Supercomputing and database architectures will often have specialized hardware that can provide true asynchrony underlying the logical asynchrony provided by this interface. In addition, asynchronous I/O should be supported by all types of files and devices in the same manner.

- Effect of Buffering

  If asynchronous I/O is performed on a file that is buffered prior to being actually written to the device, it is possible that asynchronous I/O will offer no performance advantage over normal I/O; the cycles *stolen* to perform the asynchronous I/O will be taken away from the running process and the I/O will occur at interrupt time. This potential lack of gain in performance in no way obviates the need for asynchronous I/O by realtime applications, which very often will use specialized hardware support; multiple processors; and/or unbuffered, synchronized I/O.

B.2.8.3 Memory Management

All memory management and shared memory definitions are located in the **<sys/mman.h>** header. This is for alignment with historical practice.

Memory Locking Functions

This portion of the rationale presents models, requirements, and standardization issues relevant to process memory locking.

- Models

  Realtime systems that conform to IEEE Std. 1003.1-200x are expected (and desired) to be supported on systems with demand-paged virtual memory management, non-paged swapping memory management, and physical memory systems with no memory management hardware. The general case, however, is the demand-paged, virtual memory system with each POSIX process running in a virtual address space. Note that this includes architectures where each process resides in its own virtual address space and architectures where the address space of each process is only a portion of a larger global virtual address space.

  The concept of memory locking is introduced to eliminate the indeterminacy introduced by paging and swapping, and to support an upper bound on the time required to access the memory mapped into the address space of a process. Ideally, this upper bound will be the same as the time required for the processor to access ‘‘main memory’’, including any address translation and cache miss overheads. But some implementations—primarily on mainframes—will not actually force locked pages to be loaded and held resident in main memory. Rather, they will handle locked pages so that accesses to these pages will meet the performance metrics for locked process memory in the implementation. Also, although it is not, for example, the intention that this interface, as specified, be used to lock process memory into ‘‘cache’’, it is conceivable that an implementation could support a large static RAM memory and define this as ‘‘main memory’’ and use a large[r] dynamic RAM as ‘‘backing store’’. These interfaces could then be interpreted as supporting the locking of process memory into the static RAM. Support for multiple levels of backing store would require extensions to these interfaces.

  Implementations may also use memory locking to guarantee a fixed translation between virtual and physical addresses where such is beneficial to improving determinancy for direct-to/from-process input/output. IEEE Std. 1003.1-200x does not guarantee to the application that the virtual-to-physical address translations, if such exist, are fixed, because such behavior would not be implementable on all architectures on which implementations of IEEE Std. 1003.1-200x are expected. But IEEE Std. 1003.1-200x does mandate that an implementation define, for the benefit of potential users, whether or not locking guarantees fixed translations.

  Memory locking is defined with respect to the address space of a process. Only the pages mapped into the address space of a process may be locked by the process, and when the pages are no longer mapped into the address space—for whatever reason—the locks established with respect to that address space are removed. Shared memory areas warrant special mention, as they may be mapped into more than one address space or mapped more than once into the address space of a process; locks may be established on pages within these areas with respect to several of these mappings. In such a case, the lock state of the underlying physical pages is the logical OR of the lock state with respect to each of the mappings. Only when all such locks have been removed are the shared pages considered unlocked.

In recognition of the page granularity of Memory Management Units (MMU), and in order to support locking of ranges of address space, memory locking is defined in terms of ''page'' granularity. That is, for the interfaces that support an address and size specification for the region to be locked, the address must be on a page boundary, and all pages mapped by the specified range are locked, if valid. This means that the length is implicitly rounded up to a multiple of the page size. The page size is implementation-defined and is available to applications as a compile time symbolic constant or at runtime via *sysconf*( ).

A ''real memory'' POSIX.1b implementation that has no MMU could elect not to support these interfaces, returning [ENOSYS]. But an application could easily interpret this as meaning that the implementation would unconditionally page or swap the application when such is not the case. It is the intention of IEEE Std. 1003.1-200x that such a system could define these interfaces as ''NO-OPs'', returning success without actually performing any function except for mandated argument checking.

- Requirements

For realtime applications, memory locking is generally considered to be required as part of application initialization. This locking is performed after an application has been loaded (that is, *exec*'d) and the program remains locked for its entire lifetime. But to support applications that undergo major mode changes where, in one mode, locking is required, but in another it is not, the specified interfaces allow repeated locking and unlocking of memory within the lifetime of a process.

When a realtime application locks its address space, it should not be necessary for the application to then ''touch'' all of the pages in the address space to guarantee that they are resident or else suffer potential paging delays the first time the page is referenced. Thus, IEEE Std. 1003.1-200x requires that the pages locked by the specified interfaces be resident when the locking functions return successfully.

Many architectures support system-managed stacks that grow automatically when the current extent of the stack is exceeded. A realtime application has a requirement to be able to ''preallocate'' sufficient stack space and lock it down so that it will not suffer page faults to grow the stack during critical realtime operation. There was no consensus on a portable way to specify how much stack space is needed, so IEEE Std. 1003.1-200x supports no specific interface for preallocating stack space. But an application can portably lock down a specific amount of stack space by specifying MCL_FUTURE in a call to *memlockall*( ) and then calling a dummy function that declares an automatic array of the desired size.

Memory locking for realtime applications is also generally considered to be an ''all or nothing'' proposition. That is, the entire process, or none, is locked down. But, for applications that have well-defined sections that need to be locked and others that do not, IEEE Std. 1003.1-200x supports an optional set of interfaces to lock or unlock a range of process addresses. Reasons for locking down a specific range include:

— An asynchronous event handler function that must respond to external events in a deterministic manner such that page faults cannot be tolerated

— An input/output ''buffer'' area that is the target for direct-to-process I/O, and the overhead of implicit locking and unlocking for each I/O call cannot be tolerated

Finally, locking is generally viewed as an ''application-wide'' function. That is, the application is globally aware of which regions are locked and which are not over time. This is in contrast to a function that is used temporarily within a ''third party'' library routine whose function is unknown to the application, and therefore must have no ''side effects''. The specified interfaces, therefore, do not support ''lock stacking'' or ''lock nesting'' within a process. But, for pages that are shared between processes or mapped more than once into a process address space, ''lock stacking'' is essentially mandated by the requirement that unlocking of pages that are mapped by more that one process or more than once by the same process does not affect locks established on the other mappings.

There was some support for ''lock stacking'' so that locking could be transparently used in library functions or opaque modules. But the consensus was not to burden all implementations with lock stacking (and reference counting), and an implementation option was proposed. There were strong objections to the option because applications would have to support both options in order to remain portable. The consensus was to eliminate lock stacking altogether, primarily through overwhelming support for the System V ''m[un]lock[all]'' interface on which IEEE Std. 1003.1-200x is now based.

Locks are not inherited across *fork*( )s because some systems implement *fork*( ) by creating new address spaces for the child. In such an implementation, requiring locks to be inherited would lead to new situations in which a fork would fail due to the inability of the system to lock sufficient memory to lock both the parent and the child. The consensus was that there was no benefit to such inheritance. Note that this does not mean that locks are removed when, for instance, a thread is created in the same address space.

Similarly, locks are not inherited across *exec* because some systems implement *exec* by unmapping all of the pages in the address space (which, by definition, removes the locks on these pages), and maps in pages of the *exec*'d image. In such an implementation, requiring locks to be inherited would lead to new situations in which *exec* would fail. Reporting this failure would be very cumbersome to detect in time to report to the calling process, and no appropriate mechanism exists for informing the *exec*'d process of its status.

It was determined that, if the newly loaded application required locking, it was the responsibility of that application to establish the locks. This is also in keeping with the general view that it is the responsibility of the application to be aware of all locks that are established.

There was one request to allow (not mandate) locks to be inherited across *fork*( ), and a request for a flag, MCL_INHERIT, that would specify inheritance of memory locks across *exec*s. Given the difficulties raised by this and the general lack of support for the feature in IEEE Std. 1003.1-200x, it was not added. IEEE Std. 1003.1-200x does not preclude an implementation from providing this feature for administrative purposes, such as a ''run'' command that will lock down and execute specified program. Additionally, the rationale for the objection equated *fork*( ) with creating a thread in the address space. IEEE Std. 1003.1-200x does not mandate releasing locks when creating additional threads in an existing process.

- Standardization Issues

One goal of IEEE Std. 1003.1-200x is to define a set of primitives that provide the necessary functionality for realtime applications, with consideration for the needs of other application domains where such were identified, which is based to the extent possible on existing industry practice.

The Memory Locking option is required by many realtime applications to tune performance. Such a facility is accomplished by placing constraints on the virtual memory system to limit paging of time of the process or of critical sections of the process. This facility should not be used by most non-realtime applications.

Optional features provided in IEEE Std. 1003.1-200x allow applications to lock selected address ranges with the caveat that the process is responsible for being aware of the page granularity of locking and the unnested nature of the locks.

Mapped Files Functions

The Memory Mapped Files option provides 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 by instructions as memory. The use of mapped files can significantly reduce I/O data movement since file data does not have to be copied into process data buffers as in *read*( ) and *write*( ). If more than one process maps a file, its contents are shared among them. This provides a low overhead mechanism by which processes can synchronize and communicate.

- Historical Perspective

  Realtime applications have historically been implemented using a collection of cooperating processes or tasks. In early systems, these processes ran on bare hardware (that is, without an operating system) with no memory relocation or protection. The application paradigms that arose from this environment involve the sharing of data between the processes.

  When realtime systems were implemented on top of vendor-supplied operating systems, the paradigm or performance benefits of direct access to data by multiple processes was still deemed necessary. As a result, operating systems that claim to support realtime applications must support the shared memory paradigm.

  Additionally, a number of realtime systems provide the ability to map specific sections of the physical address space into the address space of a process. This ability is required if an application is to obtain direct access to memory locations that have specific properties (for example, refresh buffers or display devices, dual ported memory locations, DMA target locations). The use of this ability is common enough to warrant some degree of standardization of its interface. This ability overlaps the general paradigm of shared memory in that, in both instances, common global objects are made addressable by individual processes or tasks.

  Finally, a number of systems also provide the ability to map process addresses to files. This provides both a general means of sharing persistent objects, and using files in a manner that optimizes memory and swapping space usage.

  Simple shared memory is clearly a special case of the more general file mapping capability. In addition, there is relatively widespread agreement and implementation of the file mapping interface. In these systems, many different types of objects can be mapped (for example, files, memory, devices, and so on) using the same mapping interfaces. This approach both minimizes interface proliferation and maximizes the generality of programs using the mapping interfaces.

- Memory Mapped Files Usage

  A memory object can be concurrently mapped into the address space of one or more processes. The *mmap*( ) and *munmap*( ) functions allow a process to manipulate their address space by mapping portions of memory objects into it and removing them from it. When multiple processes map the same memory object, they can share access to the underlying data. Implementations may restrict the size and alignment of mappings to be on *page*-size boundaries. The page size, in bytes, is the value of the system-configurable variable {PAGESIZE}, typically accessed by calling *sysconf*( ) with a *name* argument of _SC_PAGESIZE. If an implementation has no restrictions on size or alignment, it may specify a 1-byte page size.

  To map memory, a process first opens a memory object. The *ftruncate*( ) function can be used to contract or extend the size of the memory object even when the object is currently mapped. If the memory object is extended, the contents of the extended areas are zeros.

After opening a memory object, the application maps the object into its address space using the *mmap*( ) function call. Once a mapping has been established, it remains mapped until unmapped with *munmap*( ), even if the memory object is closed. The *mprotect*( ) function can be used to change the memory protections initially established by *mmap*( ).

A *close*( ) of the file descriptor, while invalidating the file descriptor itself, does not unmap any mappings established for the memory object. The address space, including all mapped regions, is inherited on *fork*( ). The entire address space is unmapped on process termination or by successful calls to any of the *exec* family of functions.

The *msync*( ) function is used to force mapped file data to permanent storage.

- Effects on Other Functions

When the Memory Mapped Files option is supported, the operation of the *open*( ), *creat*( ), and *unlink*( ) functions are a natural result of using the file system name space to map the global names for memory objects.

The *ftruncate*( ) function can be use to set the length of a sharable memory object.

The meaning of *stat*( ) fields other than the size and protection information is undefined on implementations where memory objects are not implemented using regular files. When regular files are used, the times reflect when the implementation updated the file image of the data, not when a process updated the data in memory.

The operations of *fdopen*( ), *write*( ), *read*( ), and *lseek*( ) were made unspecified for objects opened with *shm_open*( ), so that implementations that did not implement memory objects as regular files would not have to support the operation of these functions on shared memory objects.

The behavior of memory objects with respect to *close*( ), *dup*( ), *dup2*( ), *open*( ), *close*( ), *fork*( ), *_exit*( ), and the *exec* family of functions is the same as the behavior of the existing practice of the *mmap*( ) function.

A memory object can still be referenced after a close. That is, any mappings made to the file are still in effect, and reads and writes that are made to those mappings are still valid and are shared with other processes that have the same mapping. Likewise, the memory object can still be used if any references remain after its name(s) have been deleted. Any references that remain after a close must not appear to the application as file descriptors.

This is existing practice for *mmap*( ) and *close*( ). In addition, there are already mappings present (text, data, stack) that do not have open file descriptors. The text mapping in particular is considered a reference to the file containing the text. The desire was to treat all mappings by the process uniformly. Also, many modern implementations use *mmap*( ) to implement shared libraries, and it would not be desirable to keep file descriptors for each of the many libraries an application can use. It was felt there were many other existing programs that used this behavior to free a file descriptor, and thus IEEE Std. 1003.1-200x could not forbid it and still claim to be using existing practice.

For implementations that implement memory objects using memory only, memory objects will retain the memory allocated to the file after the last close and will use that same memory on the next open. Note that closing the memory object is not the same as deleting the name, since the memory object is still defined in the memory object name space.

The locks of *fcntl*( ) do not block any read or write operation, including read or write access to shared memory or mapped files. In addition, implementations that only support shared memory objects should not be required to implement record locks. The reference to *fcntl*( ) is added to make this point explicitly. The other *fcntl*( ) commands are useful with shared memory objects.

The size of pages that mapping hardware may be able to support may be a configurable value, or it may change based on hardware implementations. The addition of the _SC_PAGESIZE parameter to the *sysconf*( ) function is provided for determining the mapping page size at runtime.

Shared Memory Functions

Implementations may support the Shared Memory Objects option without supporting a general Memory Mapped Files option. 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.

- Requirements

  Shared memory is used to share data among several processes, each potentially running at different priority levels, responding to different inputs, or performing separate tasks. Shared memory is not just simply providing common access to data, it is providing the fastest possible communication between the processes. With one memory write operation, a process can pass information to as many processes as have the memory region mapped.

  As a result, shared memory provides a mechanism that can be used for all other interprocess communications facilities. It may also be used by an application for implementing more sophisticated mechanisms than semaphores and message queues.

  The need for a shared memory interface is obvious for virtual memory systems, where the operating system is directly preventing processes from accessing each other's data. However, in unprotected systems, such as those found in some embedded controllers, a shared memory interface is needed to provide a portable mechanism to allocate a region of memory to be shared and then to communicate the address of that region to other processes.

  This, then, provides the minimum functionality that a shared memory interface must have in order to support realtime applications: to allocate and name an object to be mapped into memory for potential sharing (*open*( ) or *shm_open*( )), and to make the memory object available within the address space of a process (*mmap*( )). To complete the interface, a mechanism to release the claim of a process on a shared memory object (*munmap*( )) is also needed, as well as a mechanism for deleting the name of a sharable object that was previously created (*unlink*( ) or *shm_unlink*( )).

  After a mapping has been established, an implementation should not have to provide services to maintain that mapping. All memory writes into that area will appear immediately in the memory mapping of that region by any other processes.

  Thus, requirements include:

  — Support creation of sharable memory objects and the mapping of these objects into the address space of a process.

  — Sharable memory objects should be accessed by global names accessible from all processes.

  — Support the mapping of specific sections of physical address space (such as a memory mapped device) into the address space of a process. This should not be done by the process specifying the actual address, but again by an implementation-defined global name (such as a special device name) dedicated to this purpose.

  — Support the mapping of discrete portions of these memory objects.

— Support for minimum hardware configurations that contain no physical media on which to store shared memory contents permanently.

— The ability to preallocate the entire shared memory region so that minimum hardware configurations without virtual memory support can guarantee contiguous space.

— The maximizing of performance by not requiring functionality that would require implementation interaction above creating the shared memory area and returning the mapping.

Note that the above requirements do not preclude:

— The sharable memory object from being implemented using actual files on an actual file system.

— The global name that is accessible from all processes being restricted to a file system area that is dedicated to handling shared memory.

— An implementation not providing implementation-defined global names for the purpose of physical address mapping.

- Shared Memory Objects Usage

If the Shared Memory Objects option is supported, a shared memory object may be created, or opened if it already exists, with the *shm_open*( ) function. If the shared memory object is created, it has a length of zero. The *ftruncate*( ) function can be used to set the size of the shared memory object after creation. The *shm_unlink*( ) function removes the name for a shared memory object created by *shm_open*( ).

- Shared Memory Overview

The shared memory facility defined by IEEE Std. 1003.1-200x usually results in memory locations being added to the address space of the process. The implementation returns the address of the new space to the application by means of a pointer. This works well in languages like C. However, in languages such as FORTRAN, it will not work because these languages do not have pointer types. In the bindings for such a language, either a special COMMON section will need to be defined (which is unlikely), or the binding will have to allow existing structures to be mapped. The implementation will likely have to place restrictions on the size and alignment of such structures or will have to map a suitable region of the address space of the process into the memory object, and thus into other processes. These are issues for that particular language binding. For IEEE Std. 1003.1-200x, however, the practice will not be forbidden, merely undefined.

Two potentially different name spaces are used for naming objects that may be mapped into process address spaces. When the Memory Mapped Files option is supported, files may be accessed via *open*( ). When the Shared Memory Objects option is supported, sharable memory objects that might not be files may be accessed via the *shm_open*( ) function. These options are not mutually-exclusive.

Some systems supporting the Shared Memory Objects option may choose to implement the shared memory object name space as part of the file system name space. There are several reasons for this:

— It allows applications to prevent name conflicts by use of the directory structure.

— It uses an existing mechanism for accessing global objects and prevents the creation of a new mechanism for naming global objects.

In such implementations, memory objects can be implemented using regular files, if that is what the implementation chooses. The *shm_open*( ) function can be implemented as an *open*( )

call in a fixed directory followed by a call to *fcntl*( ) to set FD_CLOEXEC. The *shm_unlink*( ) function can be implemented as an *unlink*( ) call.

On the other hand, it is also expected that small embedded systems that support the Shared Memory Objects option may wish to implement shared memory without having any file systems present. In this case, the implementations may choose to use a simple string valued name space for shared memory regions. The *shm_open*( ) function permits either type of implementation.

Some systems have hardware that supports protection of mapped data from certain classes of access and some do not. Systems that supply this functionality can support the Memory Protection option.

Some implementations restrict size, alignment, and protections to be on *page*-size boundaries. If an implementation has no restrictions on size or alignment, it may specify a 1-byte page size. Applications on implementations that do support larger pages must be cognizant of the page size since this is the alignment and protection boundary.

Simple embedded implementations may have a 1-byte page size and only support the Shared Memory Objects option. This provides simple shared memory between processes without requiring mapping hardware.

IEEE Std. 1003.1-200x is silent about how implementations that chose to implement memory objects directly would treat them with standard utilities such as *ls*, because utilities are not within the charter of IEEE Std. 1003.1-200x.

IEEE Std. 1003.1-200x specifically allows a memory object to remain referenced after a close because that is existing practice for the *mmap*( ) function.

Typed Memory Functions

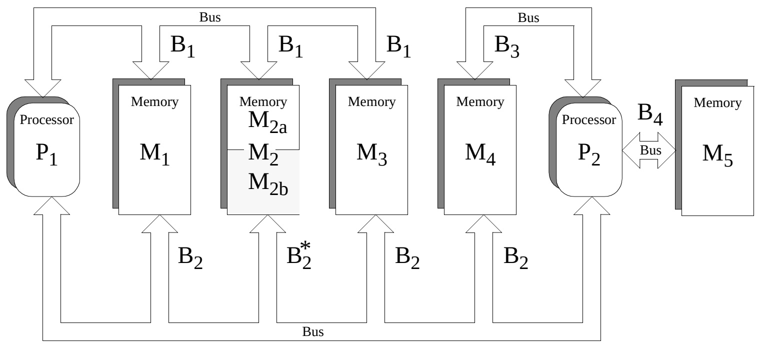
Implementations may support the Typed Memory Objects option without supporting either the Shared Memory option or the Memory Mapped Files option. Typed memory objects are pools of specialized storage, different from the main memory resource normally used by a processor to hold code and data, that can be mapped into the address space of one or more processes.

- Model

  Realtime systems conforming to one of the POSIX.13 realtime profiles are expected (and desired) to be supported on systems with more than one type or pool of memory (for example, SRAM, DRAM, ROM, EPROM, EEPROM), where each type or pool of memory may be accessible by one or more processors via one or more busses (ports). Memory mapped files, shared memory objects, and the language-specific storage allocation operators (*malloc*( ) for the ISO C standard, *new* for ANSI Ada) fail to provide application program interfaces versatile enough to allow applications to control their utilization of such diverse memory resources. The typed memory interfaces *posix_typed_mem_open*( ), *posix_mem_offset*( ), *posix_typed_mem_get_info*( ), *mmap*( ), and *munmap*( ) defined herein support the model of typed memory described below.

  For purposes of this model, a system comprises several processors (for example, P1 and P2), several physical memory pools (for example, M1, M2, M2a, M2b, M3, M4, and M5), and several busses or ‘‘ports’’ (for example, B1, B2, B3, and B4) interconnecting the various processors and memory pools in some system-specific way. Notice that some memory pools may be contained in others (for example, M2a and M2b are contained in M2).

  Figure B-1 (on page 3420) shows an example of such a model. In a system like this, an application should be able to perform the following operations:



*All addresses in pool M_2 (comprising pools M_{2a} and M_{2b}) accessible via port B_1.
Addresses in pool M_{2b} are also accessible via port B_2
Addresses in pool M_{2a} are NOT accessible via port B_2

Figure B-1 Example of a System with Typed Memory

— Typed Memory Allocation

An application should be able to allocate memory dynamically from the desired pool using the desired bus, and map it into a process' address space. For example, processor P1 can allocate some portion of memory pool M1 through port B1, treating all unmapped subareas of M1 as a heap-storage resource from which memory may be allocated. This portion of memory is mapped into the process' address space, and subsequently deallocated when unmapped from all processes.

— Using the Same Storage Region from Different Busses

An application process with a mapped region of storage that is accessed from one bus should be able to map that same storage area at another address (subject to page size restrictions detailed in *mmap*( )), to allow it to be accessed from another bus. For example, processor P1 may wish to access the same region of memory pool M2b both through ports B1 and B2.

— Sharing Typed Memory Regions

Several application processes running on the same or different processors may wish to share a particular region of a typed memory pool. Each process or processor may wish to access this region through different busses. For example, processor P1 may want to share a region of memory pool M4 with processor P2, and they may be required to use busses B2 and B3, respectively, to minimize bus contention. A problem arises here when a process allocates and maps a portion of fragmented memory and then wants to share this region of memory with another process, either in the same processor or different processors. The solution adopted is to allow the first process to find out the memory map (offsets and lengths) of all the different fragments of memory that were mapped into its address space, by repeatedly calling *posix_mem_offset*( ). Then, this process can pass the offsets and lengths obtained to the second process, which can then map the same memory fragments into its address space.

— Contiguous Allocation

The problem of finding the memory map of the different fragments of the memory pool that were mapped into logically contiguous addresses of a given process, can be solved by requesting contiguous allocation. For example, a process in P1 can allocate 10 Kbytes of physically contiguous memory from M3-B1, and obtain the offset (within pool M3) of this block of memory. Then, it can pass this offset (and the length) to a process in P2 using some interprocess communication mechanism. The second process can map the same block of memory by using the offset transferred and specifying M3-B2.

— Unallocated Mapping

Any subarea of a memory pool that is mapped to a process, either as the result of an allocation request or an explicit mapping, is normally unavailable for allocation. Special processes such as debuggers, however, may need to map large areas of a typed memory pool, yet leave those areas available for allocation.

Typed memory allocation and mapping has to coexist with storage allocation operators like *malloc*( ), but systems are free to choose how to implement this coexistence. For example, it may be system configuration-dependent if all available system memory is made part of one of the typed memory pools or if some part will be restricted to conventional allocation operators. Equally system configuration-dependent may be the availability of operators like *malloc*( ) to allocate storage from certain typed memory pools. It is not excluded to configure a system such that a given named pool, P1, is in turn split into non-overlapping named subpools. For example, M1-B1, M2-B1, and M3-B1 could also be accessed as one common pool M123-B1. A call to *malloc*( ) on P1 could work on such a larger pool while full optimization of memory usage by P1 would require typed memory allocation at the subpool level.

- Existing Practice

OS-9 provides for the naming (numbering) and prioritization of memory types by a system administrator. It then provides APIs to request memory allocation of typed (colored) memory by number, and to generate a bus address from a mapped memory address (translate). When requesting colored memory, the user can specify type 0 to signify allocation from the first available type in priority order.

HP-RT presents interfaces to map different kinds of storage regions that are visible through a VME bus, although it does not provide allocation operations. It also provides functions to perform address translation between VME addresses and virtual addresses. It represents a VME-bus unique solution to the general problem.

The PSOS approach is similar (that is, based on a pre-established mapping of bus address ranges to specific memories) with a concept of segments and regions (regions dynamically allocated from a heap which is a special segment). Therefore, PSOS does not fully address the general allocation problem either. PSOS does not have a ‘‘process’’-based model, but more of a ‘‘thread’’-only-based model of multi-tasking. So mapping to a process address space is not an issue.

QNX (a Canadian OS vendor specializing in realtime embedded systems on 80x86-based processors) uses the System V approach of opening specially named devices (shared memory segments) and using *mmap*( ) to then gain access from the process. They do not address allocation directly, but once typed shared memory can be mapped, an ‘‘allocation manager’’ process could be written to handle requests for allocation.

The System V approach also included allocation, implemented by opening yet other special ‘‘devices’’ which allocate, rather than appearing as a whole memory object.

The Orkid realtime kernel interface definition has operations to manage memory ‘‘regions’’ and ‘‘pools’’, which are areas of memory that may reflect the differing physical nature of the memory. Operations to allocate memory from these regions and pools are also provided.

- Requirements

Existing practice in SVID-derived UNIX systems relies on functionality similar to *mmap*( ) and its related interfaces to achieve mapping and allocation of typed memory. However, the issue of sharing typed memory (allocated or mapped) and the complication of multiple ports are not addressed in any consistent way by existing UNIX system practice. Part of this functionality is existing practice in specialized realtime operating systems. In order to solidify the capabilities implied by the model above, the following requirements are imposed on the interface:

— Identification of Typed Memory Pools and Ports

All processes (running in all processors) in the system shall be able to identify a particular (system configured) typed memory pool accessed through a particular (system configured) port by a name. That name shall be a member of a name space common to all these processes, but need not be the same name space as that containing ordinary file names. The association between memory pools/ports and corresponding names is typically established when the system is configured. The ''open'' operation for typed memory objects should be distinct from the *open*( ) function, for consistency with other similar services, but implementable on top of *open*( ). This implies that the handle for a typed memory object will be a file descriptor.

— Allocation and Mapping of Typed Memory

Once a typed memory object has been identified by a process, it shall be possible to both map user-selected subareas of that object into process address space and to map system-selected (that is, dynamically allocated) subareas of that object, with user-specified length, into process address space. It shall also be possible to determine the maximum length of memory allocation that may be requested from a given typed memory object.

— Sharing Typed Memory

Two or more processes shall be able to share portions of typed memory, either user-selected or dynamically allocated. This requirement applies also to dynamically allocated regions of memory that are composed of several non-contiguous pieces.

— Contiguous Allocation

For dynamic allocation, it shall be the user's option whether the system is required to allocate a contiguous subarea within the typed memory object, or whether it is permitted to allocate discontiguous fragments which appear contiguous in the process mapping. Contiguous allocation simplifies the process of sharing allocated typed memory, while discontiguous allocation allows for potentially better recovery of deallocated typed memory.

— Accessing Typed Memory Through Different Ports

Once a subarea of a typed memory object has been mapped, it shall be possible to determine the location and length corresponding to a user-selected portion of that object within the memory pool. This location and length can then be used to remap that portion of memory for access from another port. If the referenced portion of typed memory was allocated discontiguously, the length thus determined may be shorter than anticipated, and the user code shall adapt to the value returned.

— Deallocation

When a previously mapped subarea of typed memory is no longer mapped by any process in the system—as a result of a call or calls to *munmap*( )—that subarea shall become potentially reusable for dynamic allocation; actual reuse of the subarea is a function of the dynamic typed memory allocation policy.

— Unallocated Mapping

It shall be possible to map user-selected subareas of a typed memory object without marking that subarea as unavailable for allocation. This option is not the default behavior, and shall require appropriate privilege.

- Scenario

The following scenario will serve to clarify the use of the typed memory interfaces.

Process A running on P1 (see Figure B-1 (on page 3420)) wants to allocate some memory from memory pool M2, and it wants to share this portion of memory with process B running on P2. Since P2 only has access to the lower part of M2, both processes will use the memory pool named M2b which is the part of M2 that is accessible both from P1 and P2. The operations that both processes need to perform are shown below:

— Allocating Typed Memory

Process A calls *posix_typed_mem_open*( ) with the name **/typed.m2b-b1** and a *tflag* of POSIX_TYPED_MEM_ALLOCATE to get a file descriptor usable for allocating from pool M2b accessed through port B1. It then calls *mmap*( ) with this file descriptor requesting a length of 4 096 bytes. The system allocates two discontiguous blocks of sizes 1 024 and 3 072 bytes within M2b. The *mmap*( ) function returns a pointer to a 4 096 byte array in process A's logical address space, mapping the allocated blocks contiguously. Process A can then utilize the array, and store data in it.

— Determining the Location of the Allocated Blocks

Process A can determine the lengths and offsets (relative to M2b) of the two blocks allocated, by using the following procedure: First, process A calls *posix_mem_offset*( ) with the address of the first element of the array and length 4 096. Upon return, the offset and length (1 024 bytes) of the first block are returned. A second call to *posix_mem_offset*( ) is then made using the address of the first element of the array plus 1 024 (the length of the first block), and a new length of 4 096–1 024. If there were more fragments allocated, this procedure could have been continued within a loop until the offsets and lengths of all the blocks were obtained. Notice that this relatively complex procedure can be avoided if contiguous allocation is requested (by opening the typed memory object with the *tflag* POSIX_TYPED_MEM_ALLOCATE_CONTIG).

— Sharing Data Across Processes

Process A passes the two offset values and lengths obtained from the *posix_mem_offset*( ) calls to process B running on P2, via some form of interprocess communication. Process B can gain access to process A's data by calling *posix_typed_mem_open*( ) with the name **/typed.m2b-b2** and a *tflag* of zero, then using two *mmap*( ) calls on the resulting file descriptor to map the two subareas of that typed memory object to its own address space.

- Rationale for no *mem_alloc*( ) and *mem_free*( )

The standard developers had originally proposed a pair of new flags to *mmap*( ) which, when applied to a typed memory object descriptor, would cause *mmap*( ) to allocate dynamically from an unallocated and unmapped area of the typed memory object. Deallocation was similarly accomplished through the use of *munmap*( ). This was rejected by the ballot group because it excessively complicated the (already rather complex) *mmap*( ) interface and introduced semantics useful only for typed memory, to a function which must also map shared memory and files. They felt that a memory allocator should be built on top of *mmap*( ) instead of being incorporated within the same interface, much as the ISO C standard libraries build *malloc*( ) on top of the virtual memory mapping functions *brk*( ) and *sbrk*( ). This would eliminate the complicated semantics involved with unmapping only part of an allocated block of typed memory.

To attempt to achieve ballot group consensus, typed memory allocation and deallocation was first migrated from *mmap*( ) and *munmap*( ) to a pair of complementary functions modeled on the ISO C standard *malloc*( ) and *free*( ). The *mem_alloc*( ) function specified explicitly the typed memory object (typed memory pool/access port) from which allocation takes place, unlike *malloc*( ) where the memory pool and port are unspecified. The *mem_free*( ) function handled deallocation. These new semantics still met all of the requirements detailed above without modifying the behavior of *mmap*( ) except to allow it to map specified areas of typed memory objects. An implementation would have been free to implement *mem_alloc*( ) and *mem_free*( ) over *mmap*( ), through *mmap*( ), or independently but cooperating with *mmap*( ).

The ballot group was queried to see if this was an acceptable alternative, and while there was some agreement that it achieved the goal of removing the complicated semantics of allocation from the *mmap*( ) interface, several balloters realized that it just created two additional functions that behaved, in great part, like *mmap*( ). These balloters proposed an alternative which has been implemented here in place of a separate *mem_alloc*( ) and *mem_free*( ). This alternative is based on four specific suggestions:

1. The *posix_typed_mem_open*( ) function should provide a flag which specifies ''allocate on *mmap*( )'' (otherwise, *mmap*( ) just maps the underlying object). This allows things roughly similar to **/dev/zero** *versus* **/dev/swap**. Two such flags have been implemented, one of which forces contiguous allocation.

2. The *posix_mem_offset*( ) function is acceptable because it can be applied usefully to mapped objects in general. It should return the file descriptor of the underlying object.

3. The *mem_get_info*( ) function in an earlier draft should be renamed *posix_typed_mem_get_info*( ) because it is not generally applicable to memory objects. It should probably return the file descriptor's allocation attribute. We have implemented the renaming of the function, but reject having it return a piece of information which is readily known by an application without this function. Its whole purpose is to query the typed memory object for attributes that are not user-specified, but determined by the implementation.

4. There should be no separate *mem_alloc*( ) or *mem_free*( ) functions. Instead, using *mmap*( ) on a typed memory object opened with an ''allocate on *mmap*( )'' flag should be used to force allocation. These are precisely the semantics defined in the current draft.

- Rationale for no Typed Memory Access Management

The working group had originally defined an additional interface (and an additional kind of object: typed memory master) to establish and dissolve mappings to typed memory on behalf of devices or processors which were independent of the operating system and had no inherent capability to directly establish mappings on their own. This was to have provided functionality similar to device driver interfaces such as *physio*( ) and their underlying bus-specific interfaces (for example, *mballoc*( )) which serve to set up and break down DMA pathways, and derive mapped addresses for use by hardware devices and processor cards.

The ballot group felt that this was beyond the scope of POSIX.1 and its amendments. Furthermore, the removal of interrupt handling interfaces from a preceding amendment (the IEEE Std. 1003.1d-1999) during its balloting process renders these typed memory access management interfaces an incomplete solution to portable device management from a user process; it would be possible to initiate a device transfer to/from typed memory, but impossible to handle the transfer-complete interrupt in a portable way.

To achieve ballot group consensus, all references to typed memory access management capabilities were removed. The concept of portable interfaces from a device driver to both operating system and hardware is being addressed by the Uniform Driver Interface (UDI) industry forum, with formal standardization deferred until proof of concept and industry-wide acceptance and implementation.

B.2.8.4 Process Scheduling

This portion of the rationale presents models, requirements, and standardization issues relevant to process scheduling; see also Section B.2.9.4 (on page 3464).

In an operating system supporting multiple concurrent processes, the system determines the order in which processes execute to meet system-defined goals. For time-sharing systems, the goal is to enhance system throughput and promote fairness; the application is provided little or no control over this sequencing function. While this is acceptable and desirable behavior in a time-sharing system, it is inappropriate in a realtime system; realtime applications must specifically control the execution sequence of their concurrent processes in order to meet externally defined response requirements.

In IEEE Std. 1003.1-200x, the control over process sequencing is provided using a concept of scheduling policies. These policies, described in detail in this section, define the behavior of the system whenever processor resources are to be allocated to competing processes. Only the behavior of the policy is defined; conforming implementations are free to use any mechanism desired to achieve the described behavior.

- Models

  In an operating system supporting multiple concurrent processes, the system determines the order in which processes execute and might force long-running processes to yield to other processes at certain intervals. Typically, the scheduling code is executed whenever an event occurs that might alter the process to be executed next.

  The simplest scheduling strategy is a ‘‘first-in, first-out’’ (FIFO) dispatcher. Whenever a process becomes runnable, it is placed on the end of a ready list. The process at the front of the ready list is executed until it exits or becomes blocked, at which point it is removed from the list. This scheduling technique is also known as ‘‘run-to-completion’’ or ‘‘run-to-block’’.

  A natural extension to this scheduling technique is the assignment of a ‘‘non-migrating priority’’ to each process. This policy differs from strict FIFO scheduling in only one respect: whenever a process becomes runnable, it is placed at the end of the list of processes runnable at that priority level. When selecting a process to run, the system always selects the first process from the highest priority queue with a runnable process. Thus, when a process becomes unblocked, it will preempt a running process of lower priority without otherwise altering the ready list. Further, if a process elects to alter its priority, it is removed from the ready list and reinserted, using its new priority, according to the policy above.

  While the above policy might be considered unfriendly in a time-sharing environment in which multiple users require more balanced resource allocation, it could be ideal in a realtime environment for several reasons. The most important of these is that it is deterministic: the highest-priority process is always run and, among processes of equal priority, the process that has been runnable for the longest time is executed first. Because of this determinism, cooperating processes can implement more complex scheduling simply by altering their priority. For instance, if processes at a single priority were to reschedule themselves at fixed time intervals, a time-slice policy would result.

  In a dedicated operating system in which all processes are well-behaved realtime applications, non-migrating priority scheduling is sufficient. However, many existing implementations provide for more complex scheduling policies.

IEEE Std. 1003.1-200x specifies a linear scheduling model. In this model, every process in the system has a priority. The system scheduler always dispatches a process that has the highest (generally the most time-critical) priority among all runnable processes in the system. As long as there is only one such process, the dispatching policy is trivial. When multiple processes of equal priority are eligible to run, they are ordered according to a strict run-to-completion (FIFO) policy.

The priority is represented as a positive integer and is inherited from the parent process. For processes running under a fixed priority scheduling policy, the priority is never altered except by an explicit function call.

It was determined arbitrarily that larger integers correspond to ''higher priorities''.

Certain implementations might impose restrictions on the priority ranges to which processes can be assigned. There also can be restrictions on the set of policies to which processes can be set.

- Requirements

Realtime processes require that scheduling be fast and deterministic, and that it guarantees to preempt lower priority processes.

Thus, given the linear scheduling model, realtime processes require that they be run at a priority that is higher than other processes. Within this framework, realtime processes are free to yield execution resources to each other in a completely portable and implementation-defined manner.

As there is a generally perceived requirement for processes at the same priority level to share processor resources more equitably, provisions are made by providing a scheduling policy (that is, SCHED_RR) intended to provide a timeslice-like facility.

Note: The following topics assume that low numeric priority implies low scheduling criticality and *vice versa.*

- Rationale for New Interface

Realtime applications need to be able to determine when processes will run in relation to each other. It must be possible to guarantee that a critical process will run whenever it is runnable; that is, whenever it wants to for as long as it needs. SCHED_FIFO satisfies this requirement. Additionally, SCHED_RR was defined to meet a realtime requirement for a well-defined time-sharing policy for processes at the same priority.

It would be possible to use the BSD *setpriority*( ) and *getpriority*( ) functions by redefining the meaning of the ''nice'' parameter according to the scheduling policy currently in use by the process. The System V *nice*( ) interface was felt to be undesirable for realtime because it specifies an adjustment to the ''nice'' value, rather than setting it to an explicit value. Realtime applications will usually want to set priority to an explicit value. Also, System V *nice*( ) does not allow for changing the priority of another process.

With the POSIX.1b interfaces, the traditional ''nice'' value does not affect the SCHED_FIFO or SCHED_RR scheduling policies. If a ''nice'' value is supported, it is implementation-defined whether it affects the SCHED_OTHER policy.

An important aspect of IEEE Std. 1003.1-200x is the explicit description of the queuing and preemption rules. It is critical, to achieve deterministic scheduling, that such rules be stated clearly in IEEE Std. 1003.1-200x.

IEEE Std. 1003.1-200x does not address the interaction between priority and swapping. The issues involved with swapping and virtual memory paging are extremely implementation-defined and would be nearly impossible to standardize at this point. The proposed scheduling paradigm, however, fully describes the scheduling behavior of runnable processes, of which one criterion is that the working set be resident in memory. Assuming the existence of a portable interface for locking portions of a process in memory, paging behavior need not affect the scheduling of realtime processes.

IEEE Std. 1003.1-200x also does not address the priorities of ''system'' processes. In general, these processes should always execute in low-priority ranges to avoid conflict with other realtime processes. Implementations should document the priority ranges in which system processes run.

The default scheduling policy is not defined. The effect of I/O interrupts and other system processing activities is not defined. The temporary lending of priority from one process to another (such as for the purposes of affecting freeing resources) by the system is not addressed. Preemption of resources is not addressed. Restrictions on the ability of a process to affect other processes beyond a certain level (influence levels) is not addressed.

The rationale used to justify the simple time-quantum scheduler is that it is common practice to depend upon this type of scheduling to assure ''fair'' distribution of processor resources among portions of the application that must interoperate in a serial fashion. Note that IEEE Std. 1003.1-200x is silent with respect to the setting of this time quantum, or whether it is a system-wide value or a per-process value, although it appears that the prevailing realtime practice is for it to be a system-wide value.

In a system with N processes at a given priority, all processor-bound, in which the time quantum is equal for all processes at a specific priority level, the following assumptions are made of such a scheduling policy:

1. A time quantum Q exists and the current process will own control of the processor for at least a duration of Q and will have the processor for a duration of Q.
2. The Nth process at that priority will control a processor within a duration of $(N-1) \times Q$.

These assumptions are necessary to provide equal access to the processor and bounded response from the application.

The assumptions hold for the described scheduling policy only if no system overhead, such as interrupt servicing, is present. If the interrupt servicing load is non-zero, then one of the two assumptions becomes fallacious, based upon how Q is measured by the system.

If Q is measured by clock time, then the assumption that the process obtains a duration Q processor time is false if interrupt overhead exists. Indeed, a scenario can be constructed with N processes in which a single process undergoes complete processor starvation if a peripheral device, such as an analog-to-digital converter, generates significant interrupt activity periodically with a period of $N \times Q$.

If Q is measured as actual processor time, then the assumption that the Nth process runs in within the duration $(N-1) \times Q$ is false.

It should be noted that SCHED_FIFO suffers from interrupt-based delay as well. However, for SCHED_FIFO, the implied response of the system is ''as soon as possible'', so that the interrupt load for this case is a vendor selection and not a compliance issue.

With this in mind, it is necessary either to complete the definition by including bounds on the interrupt load, or to modify the assumptions that can be made about the scheduling policy.

Since the motivation of inclusion of the policy is common usage, and since current applications do not enjoy the luxury of bounded interrupt load, item (2) above is sufficient to express existing application needs and is less restrictive in the standard definition. No difference in interface is necessary.

In an implementation in which the time quantum is equal for all processes at a specific priority, our assumptions can then be restated as:

— A time quantum Q exists, and a processor-bound process will be rescheduled after a duration of, at most, Q. Time quantum Q may be defined in either wall clock time or execution time.

— In general, the Nth process of a priority level should wait no longer than $(N-1) \times Q$ time to execute, assuming no processes exist at higher priority levels.

— No process should wait indefinitely.

For implementations supporting per-process time quanta, these assumptions can be readily extended.

Sporadic Server Scheduling Policy

The sporadic server is a mechanism defined for scheduling aperiodic activities in time-critical realtime systems. This mechanism reserves a certain bounded amount of execution capacity for processing aperiodic events at a high priority level. Any aperiodic events that cannot be processed within the bounded amount of execution capacity are executed in the background at a low priority level. Thus, a certain amount of execution capacity can be guaranteed to be available for processing periodic tasks, even under burst conditions in the arrival of aperiodic processing requests (that is, a large number of requests in a short time interval). The sporadic server also simplifies the schedulability analysis of the realtime system, because it allows aperiodic processes or threads to be treated as if they were periodic. The sporadic server was first described by Sprunt, et al.

The key concept of the sporadic server is to provide and limit a certain amount of computation capacity for processing aperiodic events at their assigned normal priority, during a time interval called the *replenishment period*. Once the entity controlled by the sporadic server mechanism is initialized with its period and execution-time budget attributes, it preserves its execution capacity until an aperiodic request arrives. The request will be serviced (if there are no higher priority activities pending) as long as there is execution capacity left. If the request is completed, the actual execution time used to service it is subtracted from the capacity, and a replenishment of this amount of execution time is scheduled to happen one replenishment period after the arrival of the aperiodic request. If the request is not completed, because there is no execution capacity left, then the aperiodic process or thread is assigned a lower background priority. For each portion of consumed execution capacity the execution time used is replenished after one replenishment period. At the time of replenishment, if the sporadic server was executing at a background priority level, its priority is elevated to the normal level. Other similar replenishment policies have been defined, but the one presented here represents a compromise between efficiency and implementation complexity.

The interface that appears in this section defines a new scheduling policy for threads and processes that behaves according to the rules of the sporadic server mechanism. Scheduling attributes are defined and functions are provided to allow the user to set and get the parameters that control the scheduling behavior of this mechanism, namely the normal and low priority, the replenishment period, the maximum number of pending replenishment operations, and the initial execution-time budget.

- Scheduling Aperiodic Activities

  Virtually all realtime applications are required to process aperiodic activities. In many cases, there are tight timing constraints that the response to the aperiodic events must meet. Usual timing requirements imposed on the response to these events are:

  — The effects of an aperiodic activity on the response time of lower priority activities must be controllable and predictable.

  — The system must provide the fastest possible response time to aperiodic events.

  — It must be possible to take advantage of all the available processing bandwidth not needed by time-critical activities to enhance average-case response times to aperiodic events.

  Traditional methods for scheduling aperiodic activities are background processing, polling tasks, and direct event execution:

  — Background processing consists of assigning a very low priority to the processing of aperiodic events. It utilizes all the available bandwidth in the system that has not been consumed by higher priority threads. However, it is very difficult, or impossible, to meet requirements on average-case response time, because the aperiodic entity has to wait for the execution of all other entities which have higher priority.

  — Polling consists of creating a periodic process or thread for servicing aperiodic requests. At regular intervals, the polling entity is started and it services accumulated pending aperiodic requests. If no aperiodic requests are pending, the polling entity suspends itself until its next period. Polling allows the aperiodic requests to be processed at a higher priority level. However, worst and average-case response times of polling entities are a direct function of the polling period, and there is execution overhead for each polling period, even if no event has arrived. If the deadline of the aperiodic activity is short compared to the inter-arrival time, the polling frequency must be increased to guarantee meeting the deadline. For this case, the increase in frequency can dramatically reduce the efficiency of the system and, therefore, its capacity to meet all deadlines. Yet, polling represents a good way to handle a large class of practical problems because it preserves system predictability, and because the amortized overhead drops as load increases.

  — Direct event execution consists of executing the aperiodic events at a high fixed-priority level. Typically, the aperiodic event is processed by an interrupt service routine as soon as it arrives. This technique provides predictable response times for aperiodic events, but makes the response times of all lower priority activities completely unpredictable under burst arrival conditions. Therefore, if the density of aperiodic event arrivals is unbounded, it may be a dangerous technique for time-critical systems. Yet, for those cases in which the physics of the system imposes a bound on the event arrival rate, it is probably the most efficient technique.

  — The sporadic server scheduling algorithm combines the predictability of the polling approach with the short response times of the direct event execution. Thus, it allows systems to meet an important class of application requirements that cannot be met by using the traditional approaches. Multiple sporadic servers with different attributes can be applied to the scheduling of multiple classes of aperiodic events, each with different kinds of timing requirements, such as individual deadlines, average response times, and so on. It also has many other interesting applications for realtime, such as scheduling producer/consumer tasks in time-critical systems, limiting the effects of faults on the estimation of task execution-time requirements, and so on.

- Existing Practice

  The sporadic server has been used in different kinds of applications, including military avionics, robot control systems, industrial automation systems, and so on. There are examples of many systems that cannot be successfully scheduled using the classic approaches, such as direct event execution, or polling, and are schedulable using a sporadic server scheduler. The sporadic server algorithm itself can successfully schedule all systems scheduled with direct event execution or polling.

  The sporadic server scheduling policy has been implemented as a commercial product in the run-time system of the Verdix Ada compiler. There are also many applications that have used a much less efficient application-level sporadic server. These real-time applications would benefit from a sporadic server scheduler implemented at the scheduler level.

- Library-Level *versus* Kernel-Level Implementation

  The sporadic server interface described in this section requires the sporadic server policy to be implemented at the same level as the scheduler. This means that the process sporadic server shall be implemented at the kernel level and the thread sporadic server policy shall be implemented at the same level as the thread scheduler; that is, kernel or library level.

  In an earlier interface for the sporadic server, this mechanism was implementable at a different level than the scheduler. This feature allowed the implementer to choose between an efficient scheduler-level implementation, or a simpler user or library-level implementation. However, the working group considered that this interface made the use of sporadic servers more complex, and that library-level implementations would lack some of the important functionality of the sporadic server, namely the limitation of the actual execution time of aperiodic activities. The working group also felt that the interface described in this chapter does not preclude library-level implementations of threads intended to provide efficient low-overhead scheduling for those threads that are not scheduled under the sporadic server policy.

- Range of Scheduling Priorities

  Each of the scheduling policies supported in IEEE Std. 1003.1-200x has an associated range of priorities. The priority ranges for each policy might or might not overlap with the priority ranges of other policies. For time-critical realtime applications it is usual for periodic and aperiodic activities to be scheduled together in the same processor. Periodic activities will usually be scheduled using the SCHED_FIFO scheduling policy, while aperiodic activities may be scheduled using SCHED_SPORADIC. Since the application developer will require complete control over the relative priorities of these activities in order to meet his timing requirements, it would be desirable for the priority ranges of SCHED_FIFO and SCHED_SPORADIC to overlap completely. Therefore, although IEEE Std. 1003.1-200x does not require any particular relationship between the different priority ranges, it is recommended that these two ranges should coincide.

- Dynamically Setting the Sporadic Server Policy

  Several members of the working group requested that implementations should not be required to support dynamically setting the sporadic server scheduling policy for a thread. The reason is that this policy may have a high overhead for library-level implementations of threads, and if threads are allowed to dynamically set this policy, this overhead can be experienced even if the thread does not use that policy. By disallowing the dynamic setting of the sporadic server scheduling policy, these implementations can accomplish efficient scheduling for threads using other policies. If a strictly conforming application needs to use the sporadic server policy, and is therefore willing to pay the overhead, it must set this policy at the time of thread creation.

- Limitation of the Number of Pending Replenishments

  The number of simultaneously pending replenishment operations must be limited for each sporadic server for two reasons: an unlimited number of replenishment operations would need an unlimited number of system resources to store all the pending replenishment operations; on the other hand, in some implementations each replenishment operation will represent a source of priority inversion (just for the duration of the replenishment operation) and thus, the maximum amount of replenishments must be bounded to guarantee bounded response times. The way in which the number of replenishments is bounded is by lowering the priority of the sporadic server to *sched_ss_low_priority* when the number of pending replenishments has reached its limit. In this way, no new replenishments are scheduled until the number of pending replenishments decreases.

  In the sporadic server scheduling policy defined in IEEE Std. 1003.1-200x, the application can specify the maximum number of pending replenishment operations for a single sporadic server, by setting the value of the *sched_ss_max_repl* scheduling parameter. This value must be between one and {SS_REPL_MAX}, which is a maximum limit imposed by the implementation. The limit {SS_REPL_MAX} must be greater than or equal to {_POSIX_SS_REPL_MAX}, which is defined to be four in IEEE Std. 1003.1-200x. The minimum limit of four was chosen so that an application can at least guarantee that four different aperiodic events can be processed during each interval of length equal to the replenishment period.

B.2.8.5 Clocks and Timers

- Clocks

  IEEE Std. 1003.1-200x and the ISO C standard both define functions for obtaining system time. Implicit behind these functions is a mechanism for measuring passage of time. This specification makes this mechanism explicit and calls it a clock. The CLOCK_REALTIME clock required by IEEE Std. 1003.1-200x is a higher resolution version of the clock that maintains POSIX.1 system time. This is a ‘‘system-wide’’ clock, in that it is visible to all processes and, were it possible for multiple processes to all read the clock at the same time, they would see the same value.

  An extensible interface was defined, with the ability for implementations to define additional clocks. This was done because of the observation that many realtime platforms support multiple clocks, and it was desired to fit this model within the standard interface. But implementation-defined clocks need not represent actual hardware devices, nor are they necessarily system-wide.

- Timers

  Two timer types are required for a system to support realtime applications:

  1. One-shot

     A one-shot timer is a timer that is armed with an initial expiration time, either relative to the current time or at an absolute time (based on some timing base, such as time in seconds and nanoseconds since the Epoch). The timer expires once and then is disarmed. With the specified facilities, this is accomplished by setting the *it_value* member of the *value* argument to the desired expiration time and the *it_interval* member to zero.

  2. Periodic

     A periodic timer is a timer that is armed with an initial expiration time, again either relative or absolute, and a repetition interval. When the initial expiration occurs, the timer is reloaded with the repetition interval and continues counting. With the specified facilities, this is accomplished by setting the *it_value* member of the *value* argument to the desired initial expiration time and the *it_interval* member to the desired repetition interval.

For both of these types of timers, the time of the initial timer expiration can be specified in two ways:

1. Relative (to the current time)
2. Absolute

- Examples of Using Realtime Timers

  In the diagrams below, *S* indicates a program schedule, *R* shows a schedule method request, and *E* suggests an internal operating system event.

  — Periodic Timer: Data Logging

  During an experiment, it might be necessary to log realtime data periodically to an internal buffer or to a mass storage device. With a periodic scheduling method, a logging module can be started automatically at fixed time intervals to log the data.

  Program schedule is requested every 10 seconds.

```
        R         S         S         S         S         S
    ----+----+----+----+----+----+----+----+----+----+----+--->
        5   10   15   20   25   30   35   40   45   50   55
```

  [Time (in Seconds)]

  To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*( ) with the TIMER_ABSTIME flag reset, and with an initial expiration value and a repetition interval of 10 seconds.

  — One-shot Timer (Relative Time): Device Initialization

  In an emission test environment, large sample bags are used to capture the exhaust from a vehicle. The exhaust is purged from these bags before each and every test. With a one-shot timer, a module could initiate the purge function and then suspend itself for a predetermined period of time while the sample bags are prepared.

  Program schedule requested 20 seconds after call is issued.

```
        R                   S
    ----+----+----+----+----+----+----+----+----+----+----+--->
        5   10   15   20   25   30   35   40   45   50   55
```

  [Time (in Seconds)]

  To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*( ) with the TIMER_ABSTIME flag reset, and with an initial expiration value of 20 seconds and a repetition interval of zero.

  Note that if the program wishes merely to suspend itself for the specified interval, it could more easily use *nanosleep*( ).

  — One-shot Timer (Absolute Time): Data Transmission

The results from an experiment are often moved to a different system within a network for postprocessing or archiving. With an absolute one-shot timer, a module that moves data from a test-cell computer to a host computer can be automatically scheduled on a daily basis.

Program schedule requested for 2:30 a.m.

```
         R                                     S
-----+-----+-----+-----+-----+-----+-----+-----+-----+----->
   23:00 23:30 24:00 00:30 01:00 01:30 02:00 02:30 03:00
```

[Time of Day]

To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*( ) with the TIMER_ABSTIME flag set, and an initial expiration value equal to 2:30 a.m. of the next day.

— Periodic Timer (Relative Time): Signal Stabilization

Some measurement devices, such as emission analyzers, do not respond instantaneously to an introduced sample. With a periodic timer with a relative initial expiration time, a module that introduces a sample and records the average response could suspend itself for a predetermined period of time while the signal is stabilized and then sample at a fixed rate.

Program schedule requested 15 seconds after call is issued and every 2 seconds thereafter.

```
   R              S S S S S S S S S S S S S S S S S S S S
----+----+----+----+----+----+----+----+----+----+----+--->
    5   10   15   20   25   30   35   40   45   50   55
```

[Time (in Seconds)]

To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*( ) with TIMER_ABSTIME flag reset, and with an initial expiration value of 15 seconds and a repetition interval of 2 seconds.

— Periodic Timer (Absolute Time): Work Shift-related Processing

Resource utilization data is useful when time to perform experiments is being scheduled at a facility. With a periodic timer with an absolute initial expiration time, a module can be scheduled at the beginning of a work shift to gather resource utilization data throughout the shift. This data can be used to allocate resources effectively to minimize bottlenecks and delays and maximize facility throughput.

Program schedule requested for 2:00 a.m. and every 15 minutes thereafter.

```
         R                               S  S  S  S  S  S
-----+-----+-----+-----+-----+-----+-----+-----+-----+----->
   23:00 23:30 24:00 00:30 01:00 01:30 02:00 02:30 03:00
```

[Time of Day]

To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*( ) with TIMER_ABSTIME flag set, and with an initial expiration value equal to 2:00 a.m. and a repetition interval equal to 15 minutes.

- Relationship of Timers to Clocks

  The relationship between clocks and timers armed with an absolute time is straightforward: a timer expiration signal is requested when the associated clock reaches or exceeds the specified time. The relationship between clocks and timers armed with a relative time (an interval) is less obvious, but not unintuitive. In this case, a timer expiration signal is requested when the specified interval, *as measured by the associated clock*, has passed. For the required CLOCK_REALTIME clock, this allows timer expiration signals to be requested at specified ''wall clock'' times (absolute), or when a specified interval of ''realtime'' has passed (relative). For an implementation-defined clock—say, a process virtual time clock—timer expirations could be requested when the process has used a specified total amount of virtual time (absolute), or when it has used a specified *additional* amount of virtual time (relative).

  The interfaces also allow flexibility in the implementation of the functions. For example, an implementation could convert all absolute times to intervals by subtracting the clock value at the time of the call from the requested expiration time and ''counting down'' at the supported resolution. Or it could convert all relative times to absolute expiration time by adding in the clock value at the time of the call and comparing the clock value to the expiration time at the supported resolution. Or it might even choose to maintain absolute times as absolute and compare them to the clock value at the supported resolution for absolute timers, and maintain relative times as intervals and count them down at the resolution supported for relative timers. The choice will be driven by efficiency considerations and the underlying hardware or software clock implementation.

- Data Definitions for Clocks and Timers

  IEEE Std. 1003.1-200x uses a time representation capable of supporting nanosecond resolution timers for the following reasons:

  — To enable IEEE Std. 1003.1-200x to represent those computer systems already using nanosecond or submicrosecond resolution clocks.

  — To accommodate those per-process timers that might need nanoseconds to specify an absolute value of system-wide clocks, even though the resolution of the per-process timer may only be milliseconds, or *vice versa*.

  — Because the number of nanoseconds in a second can be represented in 32 bits.

  Time values are represented in the **timespec** structure. The *tv_sec* member is of type **time_t** so that this member is compatible with time values used by POSIX.1 functions and the ISO C standard. The *tv_nsec* member is a **signed long** in order to simplify and clarify code that decrements or finds differences of time values. Note that because 1 billion (number of nanoseconds per second) is less than half of the value representable by a signed 32-bit value, it is always possible to add two valid fractional seconds represented as integral nanoseconds without overflowing the signed 32-bit value.

  A maximum allowable resolution for the CLOCK_REALTIME clock of 20 ms (1/50 seconds) was chosen to allow line frequency clocks in European countries to be conforming. 60 Hz clocks in the U.S. will also be conforming, as will finer granularity clocks, although a Strictly Conforming Application cannot assume a granularity of less than 20 ms (1/50 seconds).

  The minimum allowable maximum time allowed for the CLOCK_REALTIME clock and the function *nanosleep*( ), and timers created with *clock_id*=CLOCK_REALTIME, is determined by the fact that the *tv_sec* member is of type **time_t**.

  IEEE Std. 1003.1-200x specifies that timer expirations shall not be delivered early, nor shall *nanosleep*( ) return early due to quantization error. IEEE Std. 1003.1-200x discusses the various implementations of *alarm*( ) in the rationale and states that implementations that do not allow alarm signals to occur early are the most appropriate, but refrained from mandating this behavior. Because of the importance of predictability to realtime applications, IEEE Std. 1003.1-200x takes a stronger stance.

The developers of IEEE Std. 1003.1-200x considered using a time representation that differs from POSIX.1b in the second 32 bit of the 64-bit value. Whereas POSIX.1b defines this field as a fractional second in nanoseconds, the other methodology defines this as a binary fraction of one second, with the radix point assumed before the most significant bit.

POSIX.1b is a software, source-level standard and most of the benefits of the alternate representation are enjoyed by hardware implementations of clocks and algorithms. It was felt that mandating this format for POSIX.1b clocks and timers would unnecessarily burden the application writer with writing, possibly non-portable, multiple precision arithmetic packages to perform conversion between binary fractions and integral units such as nanoseconds, milliseconds, and so on.

Rationale for the Monotonic Clock

For those applications that use time services to achieve realtime behavior, changing the value of the clock on which these services rely may cause erroneous timing behavior. For these applications, it is necessary to have a monotonic clock which cannot run backwards, and which has a maximum clock jump that is required to be documented by the implementation. Additionally, it is desirable (but not required by IEEE Std. 1003.1-200x) that the monotonic clock increases its value uniformly. This clock should not be affected by changes to the system time; for example, to synchronize the clock with an external source or to account for leap seconds. Such changes would cause errors in the measurement of time intervals for those time services that use the absolute value of the clock.

One could argue that by defining the behavior of time services when the value of a clock is changed, deterministic realtime behavior can be achieved. For example, one could specify that relative time services should be unaffected by changes in the value of a clock. However, there are time services that are based upon an absolute time, but that are essentially intended as relative time services. For example, *pthread_cond_timedwait*( ) uses an absolute time to allow it to wake up after the required interval despite spurious wakeups. Although sometimes the *pthread_cond_timedwait*( ) timeouts are absolute in nature, there are many occasions in which they are relative, and their absolute value is determined from the current time plus a relative time interval. In this latter case, if the clock changes while the thread is waiting, the wait interval will not be the expected length. If a *pthread_cond_timedwait*( ) function were created that would take a relative time, it would not solve the problem because to retain the intended ‘‘deadline’’ a thread would need to compensate for latency due to the spurious wakeup, and preemption between wakeup and the next wait.

The solution is to create a new monotonic clock, whose value does not change except for the regular ticking of the clock, and use this clock for implementing the various relative timeouts that appear in the different POSIX interfaces, as well as allow *pthread_cond_timedwait*( ) to choose this new clock for its timeout. A new *clock_nanosleep*( ) function is created to allow an application to take advantage of this newly defined clock. Notice that the monotonic clock may be implemented using the same hardware clock as the system clock.

Relative timeouts for *sigtimedwait*( ) and *aio_suspend*( ) have been redefined to use the monotonic clock, if present. The *alarm*( ) function has not been redefined, because the same effect but with better resolution can be achieved by creating a timer (for which the appropriate clock may be chosen).

The *pthread_cond_timedwait*( ) function has been treated in a different way, compared to other functions with absolute timeouts, because it is used to wait for an event, and thus it may have a deadline, while the other timeouts are generally used as an error recovery mechanism, and for them the use of the monotonic clock is not so important. Since the desired timeout for the *pthread_cond_timedwait*( ) function may either be a relative interval, or an absolute time of day deadline, a new initialization attribute has been created for condition variables, to specify the clock that shall be used for measuring the timeout in a call to *pthread_cond_timedwait*( ). In this way, if a relative timeout is desired, the monotonic clock will be used; if an absolute deadline is required instead, the CLOCK_REALTIME or another appropriate clock may be used. This capability has not been added to other functions with absolute timeouts because for those functions the expected use of the timeout is mostly to prevent errors, and not so often to meet precise deadlines. As a consequence, the complexity of adding this capability is not justified by its perceived application usage.

The *nanosleep*( ) function has not been modified with the introduction of the monotonic clock. Instead, a new *clock_nanosleep*( ) function has been created, in which the desired clock may be specified in the function call.

- History of Resolution Issues

  Due to the shift from relative to absolute timeouts in IEEE Std. 1003.1d-1999, the amendments to the *sem_timedwait*( ), *pthread_mutex_timedlock*( ), *mq_timedreceive*( ), and *mq_timedsend*( ) functions of that standard have been removed. Those amendments specified that CLOCK_MONOTONIC would be used for the (relative) timeouts if the Monotonic Clock option was supported.

  Having these functions continue to be tied solely to CLOCK_MONOTONIC would not work. Since the absolute value of a time value obtained from CLOCK_MONOTONIC is unspecified, under the absolute timeouts interface, applications would behave differently depending on whether the Monotonic Clock option was supported or not (because the absolute value of the clock would have different meanings in either case).

  Two options were considered:

  1. Leave the current behavior unchanged, which specifies the CLOCK_REALTIME clock for these (absolute) timeouts, to allow portability of applications between implementations supporting or not the Monotonic Clock option.

  2. Modify these functions in the way that *pthread_cond_timedwait*( ) was modified to allow a choice of clock, so that an application could use CLOCK_REALTIME when it is trying to achieve an absolute timeout and CLOCK_MONOTONIC when it is trying to achieve a relative timeout.

  It was decided that the features of CLOCK_MONOTONIC are not as critical to these functions as they are to *pthread_cond_timedwait*( ). The *pthread_cond_timedwait*( ) function is given a relative timeout; the timeout may represent a deadline for an event. When these functions are given relative timeouts, the timeouts are typically for error recovery purposes and need not be so precise.

  Therefore, it was decided that these functions should be tied to CLOCK_REALTIME and not complicated by being given a choice of clock.

Execution Time Monitoring

- Introduction

  The main goals of the execution time monitoring facilities defined in this chapter are to measure the execution time of processes and threads and to allow an application to establish CPU time limits for these entities.

  The analysis phase of time-critical realtime systems often relies on the measurement of execution times of individual threads or processes to determine whether the timing requirements will be met. Also, performance analysis techniques for soft deadline realtime systems rely heavily on the determination of these execution times. The execution time monitoring functions provide application developers with the ability to measure these execution times online and open the possibility of dynamic execution-time analysis and system reconfiguration, if required.

  The second goal of allowing an application to establish execution time limits for individual processes or threads and detecting when they overrun allows program robustness to be increased by enabling online checking of the execution times.

  If errors are detected—possibly because of erroneous program constructs, the existence of errors in the analysis phase, or a burst of event arrivals—online detection and recovery is possible in a portable way. This feature can be extremely important for many time-critical applications. Other applications require trapping CPU-time errors as a normal way to exit an algorithm; for instance, some realtime artificial intelligence applications trigger a number of independent inference processes of varying accuracy and speed, limit how long they can run, and pick the best answer available when time runs out. In many periodic systems, overrun processes are simply restarted in the next resource period, after necessary end-of-period actions have been taken. This allows algorithms that are inherently data-dependent to be made predictable.

  The interface that appears in this chapter defines a new type of clock, the CPU-time clock, which measures execution time. Each process or thread can invoke the clock and timer functions defined in POSIX.1 to use them. Functions are also provided to access the CPU-time clock of other processes or threads to enable remote monitoring of these clocks. Monitoring of threads of other processes is not supported, since these threads are not visible from outside of their own process with the interfaces defined in POSIX.1.

- Execution Time Monitoring Interface

  The clock and timer interface defined in POSIX.1 historically only defined one clock, which measures wall-clock time. The requirements for measuring execution time of processes and threads, and setting limits to their execution time by detecting when they overrun, can be accomplished with that interface if a new kind of clock is defined. These new clocks measure execution time, and one is associated with each process and with each thread. The clock functions currently defined in POSIX.1 can be used to read and set these CPU-time clocks, and timers can be created using these clocks as their timing base. These timers can then be used to send a signal when some specified execution time has been exceeded. The CPU-time clocks of each process or thread can be accessed by using the symbols CLOCK_PROCESS_CPUTIME_ID or CLOCK_THREAD_CPUTIME_ID.

  The clock and timer interface defined in POSIX.1 and extended with the new kind of CPU-time clock would only allow processes or threads to access their own CPU-time clocks. However, many realtime systems require the possibility of monitoring the execution time of processes or threads from independent monitoring entities. In order to allow applications to construct independent monitoring entities that do not require cooperation from or modification of the monitored entities, two functions have been added: *clock_getcpuclockid*( ), for accessing CPU-time clocks of other processes, and *pthread_getcpuclockid*( ), for accessing CPU-time clocks of other threads. These functions return the clock identifier associated with the process or thread specified in the call. These clock IDs can then be used in the rest of the clock function calls.

The clocks accessed through these functions could also be used as a timing base for the creation of timers, thereby allowing independent monitoring entities to limit the CPU-time consumed by other entities. However, this possibility would imply additional complexity and overhead because of the need to maintain a timer queue for each process or thread, to store the different expiration times associated with timers created by different processes or threads. The working group decided this additional overhead was not justified by application requirements. Therefore, creation of timers attached to the CPU-time clocks of other processes or threads has been specified as implementation-defined.

- Overhead Considerations

  The measurement of execution time may introduce additional overhead in the thread scheduling, because of the need to keep track of the time consumed by each of these entities. In library-level implementations of threads, the efficiency of scheduling could be somehow compromised because of the need to make a kernel call, at each context switch, to read the process CPU-time clock. Consequently, a thread creation attribute called *cpu-clock-requirement* was defined, to allow threads to disconnect their respective CPU-time clocks. However, the Ballot Group considered that this attribute itself introduced some overhead, and that in current implementations it was not worth the effort. Therefore, the attribute was deleted, and thus thread CPU-time clocks are required for all threads if the Thread CPU-Time Clocks option is supported.

- Accuracy of CPU-time Clocks

  The mechanism used to measure the execution time of processes and threads is specified in IEEE Std. 1003.1-200x as implementation-defined. The reason for this is that both the underlying hardware and the implementation architecture have a very strong influence on the accuracy achievable for measuring CPU time. For some implementations, the specification of strict accuracy requirements would represent very large overheads, or even the impossibility of being implemented.

  Since the mechanism for measuring execution time is implementation-defined, realtime applications will be able to take advantage of accurate implementations using a portable interface. Of course, strictly conforming applications cannot rely on any particular degree of accuracy, in the same way as they cannot rely on a very accurate measurement of wall clock time. There will always exist applications whose accuracy or efficiency requirements on the implementation are more rigid than the values defined in IEEE Std. 1003.1-200x or any other standard.

  In any case, there is a minimum set of characteristics that realtime applications would expect from most implementations. One such characteristic is that the sum of all the execution times of all the threads in a process equals the process execution time, when no CPU-time clocks are disabled. This need not always be the case because implementations may differ in how they account for time during context switches. Another characteristic is that the sum of the execution times of all processes in a system equals the number of processors, multiplied by the elapsed time, assuming that no processor is idle during that elapsed time. However, in some systems it might not be possible to relate CPU-time to elapsed time. For example, in a heterogeneous multi-processor system in which each processor runs at a different speed, an implementation may choose to define each ‘‘second’’ of CPU-time to be a certain number of ‘‘cycles’’ that a CPU has executed.

- Existing Practice

  Measuring and limiting the execution time of each concurrent activity are common features of most industrial implementations of realtime systems. Almost all critical realtime systems are currently built upon a cyclic executive. With this approach, a regular timer interrupt kicks off the next sequence of computations. It also checks that the current sequence has completed. If it has not, then some error recovery action can be undertaken (or at least an overrun is avoided). Current software engineering principles and the increasing complexity of software are driving application developers to implement these systems on multi-threaded or multi-process operating systems. Therefore, if a POSIX operating system is to be used for this type of application, then it must offer the same level of protection.

  Execution time clocks are also common in most UNIX implementations, although these clocks usually have requirements different from those of realtime applications. The POSIX.1 *times*( ) function supports the measurement of the execution time of the calling process, and its terminated child processes. This execution time is measured in clock ticks and is supplied as two different values with the user and system execution times, respectively. BSD supports the function *getrusage*( ), which allows the calling process to get information about the resources used by itself and/or all of its terminated child processes. The resource usage includes user and system CPU time. Some UNIX systems have options to specify high resolution (up to one microsecond) CPU time clocks using the *times*( ) or the *getrusage*( ) functions.

  The *times*( ) and *getrusage*( ) interfaces do not meet important realtime requirements, such as the possibility of monitoring execution time from a different process or thread, or the possibility of detecting an execution time overrun. The latter requirement is supported in some UNIX implementations that are able to send a signal when the execution time of a process has exceeded some specified value. For example, BSD defines the functions *getitimer*( ) and *setitimer*( ), which can operate either on a realtime clock (wall-clock), or on virtual-time or profile-time clocks which measure CPU time in two different ways. These functions do not support access to the execution time of other processes.

  IBM's MVS operating system supports per-process and per-thread execution time clocks. It also supports limiting the execution time of a given process.

  Given all this existing practice, the working group considered that the POSIX.1 clocks and timers interface was appropriate to meet most of the requirements that realtime applications have for execution time clocks. Functions were added to get the CPU time clock IDs, and to allow/disallow the thread CPU time clocks (in order to preserve the efficiency of some implementations of threads).

- Clock Constants

  The definition of the manifest constants CLOCK_PROCESS_CPUTIME_ID and CLOCK_THREAD_CPUTIME_ID allows processes or threads, respectively, to access their own execution-time clocks. However, given a process or thread, access to its own execution-time clock is also possible if the clock ID of this clock is obtained through a call to *clock_getcpuclockid*( ) or *pthread_getcpuclockid*( ). Therefore, these constants are not necessary and could be deleted to make the interface simpler. Their existence saves one system call in the first access to the CPU-time clock of each process or thread. The working group considered this issue and decided to leave the constants in IEEE Std. 1003.1-200x because they are closer to the POSIX.1b use of clock identifiers.

- Library Implementations of Threads

  In library implementations of threads, kernel entities and library threads can coexist. In this case, if the CPU-time clocks are supported, most of the clock and timer functions will need to have two implementations: one in the thread library, and one in the system calls library. The main difference between these two implementations is that the thread library implementation will have to deal with clocks and timers that reside in the thread space, while the kernel implementation will operate on timers and clocks that reside in kernel space. In the library implementation, if the clock ID refers to a clock that resides in the kernel, a kernel call will have to be made. The correct version of the function can be chosen by specifying the appropriate order for the libraries during the link process.

- History of Resolution Issues: Deletion of the *enable* Attribute

  In the draft corresponding to the first balloting round, CPU-time clocks had an attribute called *enable*. This attribute was introduced by the working group to allow implementations to avoid the overhead of measuring execution time for those processes or threads for which this measurement was not required. However, the *enable* attribute got several ballot objections. The main reason was that processes are already required to measure execution time by the POSIX.1 *times*( ) function. Consequently, the enable attribute was considered unnecessary, and was deleted from the draft.

Rationale Relating to Timeouts

- Requirements for Timeouts

  Realtime systems which must operate reliably over extended periods without human intervention are characteristic in embedded applications such as avionics, machine control, and space exploration, as well as more mundane applications such as cable TV, security systems, and plant automation. A multi-tasking paradigm, in which many independent and/or cooperating software functions relinquish the processor(s) while waiting for a specific stimulus, resource, condition, or operation completion, is very useful in producing well engineered programs for such systems. For such systems to be robust and fault-tolerant, expected occurrences that are unduly delayed or that never occur must be detected so that appropriate recovery actions may be taken. This is difficult if there is no way for a task to regain control of a processor once it has relinquished control (blocked) awaiting an occurrence which, perhaps because of corrupted code, hardware malfunction, or latent software bugs, will not happen when expected. Therefore, the common practice in realtime operating systems is to provide a capability to timeout such blocking services. Although there are several methods to achieve this already defined by POSIX, none are as reliable or efficient as initiating a timeout simultaneously with initiating a blocking service. This is especially critical in hard-realtime embedded systems because the processors typically have little time reserve, and allowed fault recovery times are measured in milliseconds rather than seconds.

  The working group largely agreed that such timeouts were necessary and ought to become part of IEEE Std. 1003.1-200x, particularly vendors of realtime operating systems whose customers had already expressed a strong need for timeouts. There was some resistance to inclusion of timeouts in IEEE Std. 1003.1-200x because the desired effect, fault tolerance, could, in theory, be achieved using existing facilities and alternative software designs, but there was no compelling evidence that realtime system designers would embrace such designs at the sacrifice of performance and/or simplicity.

- Which Services should be Timed Out?

  Originally, the working group considered the prospect of providing timeouts on all blocking services, including those currently existing in POSIX.1, POSIX.1b, and POSIX.1c, and future interfaces to be defined by other working groups, as sort of a general policy. This was rather quickly rejected because of the scope of such a change, and the fact that many of those services would not normally be used in a realtime context. More traditional timesharing solutions to timeout would suffice for most of the POSIX.1 interfaces, while others had asynchronous alternatives which, while more complex to utilize, would be adequate for some realtime and all non-realtime applications.

The list of potential candidates for timeouts was narrowed to the following for further consideration:

— POSIX.1b
  — *sem_wait*( )
  — *mq_receive*( )
  — *mq_send*( )
  — *lio_listio*( )
  — *aio_suspend*( )
  — *sigwait*( ) (timeout already implemented by *sigtimedwait*( ))
— POSIX.1c
  — *pthread_mutex_lock*( )
  — *pthread_join*( )
  — *pthread_cond_wait*( ) (timeout already implemented by *pthread_cond_timedwait*( ))
— POSIX.1
  — *read*( )
  — *write*( )

After further review by the working group, the *lio_listio*( ), *read*( ), and *write*( ) functions (all forms of blocking synchronous I/O) were eliminated from the list because of the following:

— Asynchronous alternatives exist
— Timeouts can be implemented, albeit non-portably, in device drivers
— A strong desire not to introduce modifications to POSIX.1 interfaces

The working group ultimately rejected *pthread_join*( ) since both that interface and a timed variant of that interface are non-minimal and may be implemented as a library function. See below for a library implementation of *pthread_join*( ).

Thus, there was a consensus among the working group members to add timeouts to 4 of the remaining 5 functions (the timeout for *aio_suspend*( ) was ultimately added directly to POSIX.1b, while the others were added by POSIX.1d). However, *pthread_mutex_lock*( ) remained contentious.

Many feel that *pthread_mutex_lock*( ) falls into the same class as the other functions; that is, it is desirable to timeout a mutex lock because a mutex may fail to be unlocked due to errant or corrupted code in a critical section (looping or branching outside of the unlock code), and therefore is equally in need of a reliable, simple, and efficient timeout. In fact, since mutexes are intended to guard small critical sections, most *pthread_mutex_lock*( ) calls would be expected to obtain the lock without blocking nor utilizing any kernel service, even in implementations of threads with global contention scope; the timeout alternative need only be considered after it is determined that the thread must block.

Those opposed to timing out mutexes feel that the very simplicity of the mutex is compromised by adding a timeout semantic, and that to do so is senseless. They claim that if a timed mutex is really deemed useful by a particular application, then it can be constructed from the facilities already in POSIX.1b and POSIX.1c. The following two C-language library implementations of mutex locking with timeout represent the solutions offered (in both implementations, the timeout parameter is specified as absolute time, not relative time as in the proposed POSIX.1c interfaces).

- Spinlock Implementation

```
#include <pthread.h>
#include <time.h>
#include <errno.h>

int pthread_mutex_timedlock(pthread_mutex_t *mutex,
        const struct timespec *timeout)
    {
    struct timespec timenow;

    while (pthread_mutex_trylock(mutex) == EBUSY)
        {
        clock_gettime(CLOCK_REALTIME, &timenow);
        if (timespec_cmp(&timenow,timeout) >= 0)
            {
            return ETIMEDOUT;
        }
        pthread_yield();
        }
    return 0;
    }
```

The Spinlock implementation is generally unsuitable for any application using priority-based thread scheduling policies such as SCHED_FIFO or SCHED_RR, since the mutex could currently be held by a thread of lower priority within the same allocation domain, but since the waiting thread never blocks, only threads of equal or higher priority will ever run, and the mutex cannot be unlocked. Setting priority inheritance or priority ceiling protocol on the mutex does not solve this problem, since the priority of a mutex owning thread is only boosted if higher priority threads are blocked waiting for the mutex; clearly not the case for this spinlock.

- Condition Wait Implementation

```
#include <pthread.h>
#include <time.h>
#include <errno.h>

struct timed_mutex
    {
    int locked;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    };
typedef struct timed_mutex timed_mutex_t;

int timed_mutex_lock(timed_mutex_t *tm,
        const struct timespec *timeout)
    {
    int timedout=FALSE;
    int error_status;
```

```
        pthread_mutex_lock(&tm->mutex);
        while (tm->locked && !timedout)
            {
            if ((error_status=pthread_cond_timedwait(&tm->cond,
                &tm->mutex,
                timeout))!=0)
            {
            if (error_status==ETIMEDOUT) timedout = TRUE;
            }
        }
        if(timedout)
            {
            pthread_mutex_unlock(&tm->mutex);
            return ETIMEDOUT;
            }
        else
            {
            tm->locked = TRUE;
            pthread_mutex_unlock(&tm->mutex);
            return 0;
            }
        }
    void timed_mutex_unlock(timed_mutex_t *tm)
        {
        pthread_mutex_lock(&tm->mutex); / for case assignment not atomic /
        tm->locked = FALSE;
        pthread_mutex_unlock(&tm->mutex);
        pthread_cond_signal(&tm->cond);
        }
```

The Condition Wait implementation effectively substitutes the *pthread_cond_timedwait*( ) function (which is currently timed out) for the desired *pthread_mutex_timedlock*( ). Since waits on condition variables currently do not include protocols which avoid priority inversion, this method is generally unsuitable for realtime applications because it does not provide the same priority inversion protection as the untimed *pthread_mutex_lock*( ). Also, for any given implementations of the current mutex and condition variable primitives, this library implementation has a performance cost at least 2.5 times that of the untimed *pthread_mutex_lock*( ) even in the case where the timed mutex is readily locked without blocking (the interfaces required for this case are shown in bold). Even in uniprocessors or where assignment is atomic, at least an additional *pthread_cond_signal*( ) is required. *pthread_mutex_timedlock*( ) could be implemented at effectively no performance penalty in this case because the timeout parameters need only be considered after it is determined that the mutex cannot be locked immediately.

Thus it has not yet been shown that the full semantics of mutex locking with timeout can be efficiently and reliably achieved using existing interfaces. Even if the existence of an acceptable library implementation were proven, it is difficult to justify why the interface itself should not be made portable, especially considering approval for the other four timeouts.

- Rationale for Library Implementation of *pthread_timedjoin*( )

Library implementation of *pthread_timedjoin*( ):

```
/*
 * Construct a thread variety entirely from existing functions
 * with which a join can be done, allowing the join to time out.
 */
#include <pthread.h>
#include <time.h>

struct timed_thread {
    pthread_t t;
    pthread_mutex_t m;
    int exiting;
    pthread_cond_t exit_c;
    void *(*start_routine)(void *arg);
    void *arg;
    void *status;
};

typedef struct timed_thread *timed_thread_t;
static pthread_key_t timed_thread_key;
static pthread_once_t timed_thread_once = PTHREAD_ONCE_INIT;

static void timed_thread_init()
{
    pthread_key_create(&timed_thread_key, NULL);
}

static void *timed_thread_start_routine(void *args)

/*
 * Routine to establish thread-specific data value and run the actual
 * thread start routine which was supplied to timed_thread_create().
 */
{
    timed_thread_t tt = (timed_thread_t) args;

    pthread_once(&timed_thread_once, timed_thread_init);
    pthread_setspecific(timed_thread_key, (void *)tt);
    timed_thread_exit((tt->start_routine)(tt->arg));
}

int timed_thread_create(timed_thread_t ttp, const pthread_attr_t *attr,
    void *(*start_routine)(void *), void *arg)

/*
 * Allocate a thread which can be used with timed_thread_join().
 */
{
    timed_thread_t tt;
    int result;

    tt = (timed_thread_t) malloc(sizeof(struct timed_thread));
    pthread_mutex_init(&tt->m,NULL);
    tt->exiting = FALSE;
    pthread_cond_init(&tt->exit_c,NULL);
    tt->start_routine = start_routine;
```

```
        tt->arg = arg;
        tt->status = NULL;
        if ((result = pthread_create(&tt->t, attr,
            timed_thread_start_routine, (void *)tt)) != 0) {
            free(tt);
            return result;
        }
        pthread_detach(tt->t);
        ttp = tt;
        return 0;
    }
    int timed_thread_join(timed_thread_t tt,
        struct timespec *timeout,
        void **status)
    {
        int result;
        pthread_mutex_lock(&tt->m);
        result = 0;
        /*
         * Wait until the thread announces that it is exiting,
         * or until timeout.
         */
        while (result == 0 && ! tt->exiting) {
            result = pthread_cond_timedwait(&tt->exit_c, &tt->m, timeout);
        }
        pthread_mutex_unlock(&tt->m);
        if (result == 0 && tt->exiting) {
            *status = tt->status;
            free((void *)tt);
            return result;
        }
        return result;
    }
    void timed_thread_exit(void *status)
    {
        timed_thread_t tt;
        void *specific;
        if ((specific=pthread_getspecific(timed_thread_key)) == NULL){
            /*
             * Handle cases which won't happen with correct usage.
             */
            pthread_exit( NULL);
        }
        tt = (timed_thread_t) specific;
        pthread_mutex_lock(&tt->m);
        /*
         * Tell a joiner that we're exiting.
         */
        tt->status = status;
```

```
        tt->exiting = TRUE;
        pthread_cond_signal(&tt->exit_c);
        pthread_mutex_unlock(&tt->m);
        /*
         * Call pthread exit() to call destructors and really
         * exit the thread.
         */
        pthread_exit(NULL);
    }
```

The *pthread_join*( ) C-language example shown above demonstrates that it is possible, using existing pthread facilities, to construct a variety of thread which allows for joining such a thread, but which allows the join operation to time out. It does this by using a *pthread_cond_timedwait*( ) to wait for the thread to exit. A **timed_thread_t** descriptor structure is used to pass parameters from the creating thread to the created thread, and from the exiting thread to the joining thread. This implementation is roughly equivalent to what a normal *pthread_join*( ) implementation would do, with the single change being that *pthread_cond_timedwait*( ) is used in place of a simple *pthread_cond_wait*( ).

Since it is possible to implement such a facility entirely from existing pthread interfaces, and with roughly equal efficiency and complexity to an implementation which would be provided directly by a pthreads implementation, it was the consensus of the working group members that any *pthread_timedjoin*( ) facility would be unnecessary, and should not be provided.

- Form of the Timeout Interfaces

  The working group considered a number of alternative ways to add timeouts to blocking services. At first, a system interface which would specify a one-shot or persistent timeout to be applied to subsequent blocking services invoked by the calling process or thread was considered because it allowed all blocking services to be timed out in a uniform manner with a single additional interface; this was rather quickly rejected because it could easily result in the wrong services being timed out.

  It was suggested that a timeout value might be specified as an attribute of the object (semaphore, mutex, message queue, and so on), but there was no consensus on this, either on a case-by-case basis or for all timeouts.

  Looking at the two existing timeouts for blocking services indicates that the working group members favor a separate interface for the timed version of a function. However, *pthread_cond_timedwait*( ) utilizes an absolute timeout value while *sigtimedwait*( ) uses a relative timeout value. The working group members agreed that relative timeout values are appropriate where the timeout mechanism's primary use was to deal with an unexpected or error situation, but they are inappropriate when the timeout must expire at a particular time, or before a specific deadline. For the timeouts being introduced in IEEE Std. 1003.1-200x, the working group considered allowing both relative and absolute timeouts as is done with POSIX.1b timers, but ultimately favored the simpler absolute timeout form.

  An absolute time measure can be easily implemented on top of an interface that specifies relative time, by reading the clock, calculating the difference between the current time and the desired wake-up time, and issuing a relative timeout call. But there is a race condition with this approach because the thread could be preempted after reading the clock, but before making the timed out call; in this case, the thread would be awakened later than it should and, thus, if the wake up time represented a deadline, it would miss it.

There is also a race condition when trying to build a relative timeout on top of an interface that specifies absolute timeouts. In this case, we would have to read the clock to calculate the absolute wake-up time as the sum of the current time plus the relative timeout interval. In this case, if the thread is preempted after reading the clock but before making the timed out call, the thread would be awakened earlier than desired.

But the race condition with the absolute timeouts interface is not as bad as the one that happens with the relative timeout interface, because there are simple workarounds. For the absolute timeouts interface, if the timing requirement is a deadline, we can still meet this deadline because the thread woke up earlier than the deadline. If the timeout is just used as an error recovery mechanism, the precision of timing is not really important. If the timing requirement is that between actions A and B a minimum interval of time must elapse, we can safely use the absolute timeout interface by reading the clock after action A has been started. It could be argued that, since the call with the absolute timeout is atomic from the application point of view, it is not possible to read the clock after action A, if this action is part of the timed out call. But if we look at the nature of the calls for which we specify timeouts (locking a mutex, waiting for a semaphore, waiting for a message, or waiting until there is space in a message queue), the timeouts that an application would build on these actions would not be triggered by these actions themselves, but by some other external action. For example, if we want to wait for a message to arrive to a message queue, and wait for at least 20 milliseconds, this time interval would start to be counted from some event that would trigger both the action that produces the message, as well as the action that waits for the message to arrive, and not by the wait-for-message operation itself. In this case, we could use the workaround proposed above.

For these reasons, the absolute timeout is preferred over the relative timeout interface.

B.2.9 Threads

Threads will normally be more expensive than subroutines (or functions, routines, and so on) if specialized hardware support is not provided. Nevertheless, threads should be sufficiently efficient to encourage their use as a medium to fine-grained structuring mechanism for parallelism in an application. Structuring an application using threads then allows it to take immediate advantage of any underlying parallelism available in the host environment. This means implementors are encouraged to optimize for fast execution at the possible expense of efficient utilization of storage. For example, a common thread creation technique is to cache appropriate thread data structures. That is, rather than releasing system resources, the implementation retains these resources and reuses them when the program next asks to create a new thread. If this reuse of thread resources is to be possible, there has to be very little unique state associated with each thread, because any such state has to be reset when the thread is reused.

Thread Creation Attributes

Attributes objects are provided for threads, mutexes, and condition variables as a mechanism to support probable future standardization in these areas without requiring that the interface itself be changed. Attributes objects provide clean isolation of the configurable aspects of threads. For example, ‘‘stack size’’ is an important attribute of a thread, but it cannot be expressed portably. When porting a threaded program, stack sizes often need to be adjusted. The use of attributes objects can help by allowing the changes to be isolated in a single place, rather than being spread across every instance of thread creation.

Attributes objects can be used to set up *classes* of threads with similar attributes; for example, ‘‘threads with large stacks and high priority’’ or ‘‘threads with minimal stacks’’. These classes can be defined in a single place and then referenced wherever threads need to be created.

Changes to ''class'' decisions become straightforward, and detailed analysis of each *pthread_create*( ) call is not required.

The attributes objects are defined as opaque types as an aid to extensibility. If these objects had 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 will 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.

Because assignment is not necessarily defined on a given opaque type, implementation-dependent default values cannot be defined in a portable way. The solution to this problem is to allow attribute 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 bit-vector 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**( ) or *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 bits or 4 bits, but priority requires 5 bits 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 IEEE Std. 1003.1-200x has to be done with care so as not to affect binary-compatibility.

Attribute 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.

Thread Implementation Models

There are various thread implementation models. At one end of the spectrum is the ''library-thread model''. In such a model, the threads of a process are not visible to the operating system kernel, and the threads are not kernel scheduled entities. The process is the only kernel scheduled entity. The process is scheduled onto the processor by the kernel according to the scheduling attributes of the process. The threads are scheduled onto the single kernel scheduled entity (the process) by the runtime library according to the scheduling attributes of the threads. A problem with this model is that it constrains concurrency. Since there is only one kernel scheduled entity (namely, the process), only one thread per process can execute at a time. If the thread that is executing blocks on I/O, then the whole process blocks.

At the other end of the spectrum is the ''kernel-thread model''. In this model, all threads are visible to the operating system kernel. Thus, all threads are kernel scheduled entities, and all threads can concurrently execute. The threads are scheduled onto processors by the kernel according to the scheduling attributes of the threads. The drawback to this model is that the creation and management of the threads entails operating system calls, as opposed to subroutine calls, which makes kernel threads heavier weight than library threads.

Hybrids of these two models are common. A hybrid model offers the speed of library threads and the concurrency of kernel threads. In hybrid models, a process has some (relatively small) number of kernel scheduled entities associated with it. It also has a potentially much larger number of library threads associated with it. Some library threads may be bound to kernel scheduled entities, while the other library threads are multiplexed onto the remaining kernel scheduled entities. There are two levels of thread scheduling:

1. The runtime library manages the scheduling of (unbound) library threads onto kernel scheduled entities.
2. The kernel manages the scheduling of kernel scheduled entities onto processors.

For this reason, a hybrid model is referred to as a *two-level threads scheduling model*. In this model, the process can have multiple concurrently executing threads; specifically, it can have as many concurrently executing threads as it has kernel scheduled entities.

Thread-Specific Data

Many applications require that a certain amount of context be maintained on a per-thread basis across procedure calls. A common example is a multi-threaded library routine that allocates resources from a common pool and maintains an active resource list for each thread. The thread-specific data interface provided to meet these needs may be viewed as a two-dimensional array of values with keys serving as the row index and thread IDs as the column index (although the implementation need not work this way).

- Models

  Three possible thread-specific data models were considered:

  1. No Explicit Support

A standard thread-specific data interface is not strictly necessary to support applications that require per-thread context. One could, for example, provide a hash function that converted a **pthread_t** into an integer value that could then be used to index into a global array of per-thread data pointers. This hash function, in conjunction with *pthread_self*( ), would be all the interface required to support a mechanism of this sort. Unfortunately, this technique is cumbersome. It can lead to duplicated code as each set of cooperating modules implements their own per-thread data management schemes.

2. Single (**void** *) Pointer

   Another technique would be to provide a single word of per-thread storage and a pair of functions to fetch and store the value of this word. The word could then hold a pointer to a block of per-thread memory. The allocation, partitioning, and general use of this memory would be entirely up to the application. Although this method is not as problematic as technique 1, it suffers from interoperability problems. For example, all modules using the per-thread pointer would have to agree on a common usage protocol.

3. Key / Value Mechanism

   This method associates an opaque key (for example, stored in a variable of type **pthread_key_t**) with each per-thread datum. These keys play the role of identifiers for per-thread data. This technique is the most generic and avoids the problems noted above, albeit at the cost of some complexity.

The primary advantage of the third model is its information hiding properties. Modules using this model are free to create and use their own key(s) independent of all other such usage, whereas the other models require that all modules that use thread-specific context explicitly cooperate with all other such modules. The data-independence provided by the third model is worth the additional interface.

- Requirements

It is important that it be possible to implement the thread-specific data interface without the use of thread private memory. To do otherwise would increase the weight of each thread, thereby limiting the range of applications for which the threads interfaces provided by IEEE Std. 1003.1-200x is appropriate.

The values that one binds to the key via *pthread_setspecific*( ) may, in fact, be pointers to shared storage locations available to all threads. It is only the key/value bindings that are maintained on a per-thread basis, and these can be kept in any portion of the address space that is reserved for use by the calling thread (for example, on the stack). Thus, no per-thread MMU state is required to implement the interface. On the other hand, there is nothing in the interface specification to preclude the use of a per-thread MMU state if it is available (for example, the key values returned by *pthread_key_create*( ) could be thread private memory addresses).

- Standardization Issues

Thread-specific data is a requirement for a usable thread interface. The binding described in this section provides a portable thread-specific data mechanism for languages that do not directly support a thread-specific storage class. A binding to IEEE Std. 1003.1-200x for a language that does include such a storage class need not provide this specific interface.

If a language were to include the notion of thread-specific storage, it would be desirable (but *not* required) to provide an implementation of the pthreads thread-specific data interface based on the language feature. For example, assume that a compiler for a C-like language supports a *private* storage class that provides thread-specific storage. Something similar to the following macros might be used to effect a compatible implementation:

```
#define pthread_key_t                     private void *
#define pthread_key_create(key)           /* no-op */
#define pthread_setspecific(key,value)    (key)=(value)
#define pthread_getspecific(key)          (key)
```

Note: For the sake of clarity, this example ignores destructor functions. A correct implementation would have to support them.

Barriers

- Background

Barriers are typically used in parallel DO/FOR loops to ensure that all threads have reached a particular stage in a parallel computation before allowing any to proceed to the next stage. Highly efficient implementation is possible on machines which support a ''Fetch and Add'' operation as described in the referenced Almasi and Gottlieb (1989).

The use of return value PTHREAD_BARRIER_SERIAL_THREAD is shown in the following example:

```
if ( (status=pthread_barrier_wait(&barrier)) ==
    PTHREAD_BARRIER_SERIAL_THREAD) {
    ...serial section
    }
        else if (status != 0) {
        ...error processing
    }
status=pthread_barrier_wait(&barrier);
...
```

This behavior allows a serial section of code to be executed by one thread as soon as all threads reach the first barrier. The second barrier prevents the other threads from proceeding until the serial section being executed by the one thread has completed.

Although barriers can be implemented with mutexes and condition variables, the referenced Almasi and Gottlieb (1989) provides ample illustration that such implementations are significantly less efficient than is possible. While the relative efficiency of barriers may well vary by implementation, it is important that they be recognized in the IEEE Std. 1003.1-200x to facilitate application portability while providing the necessary freedom to implementors.

- Lack of Timeout Feature

Alternate versions of most blocking routines have been provided to support watchdog timeouts. No alternate interface of this sort has been provided for barrier waits for the following reasons:

  - Multiple threads may use different timeout values, some of which may be indefinite. It is not clear which threads should break through the barrier with a timeout error if and when these timeouts expire.

  - The barrier may become unusable once a thread breaks out of a *pthread_barrier_wait*( ) with a timeout error. There is, in general, no way to guarantee the consistency of a barrier's internal data structures once a thread has timed out of a *pthread_barrier_wait*( ). Even the inclusion of a special barrier reinitialization function would not help much since it is not clear how this function would affect the behavior of threads that reach the barrier between the original timeout and the call to the reinitialization function.

Spin Locks

- Background

  Spin locks represent an extremely low-level synchronization mechanism suitable primarily for use on shared memory multi-processors. It is typically an atomically modified Boolean value that is set to one when the lock is held and to zero when the lock is freed.

  When a caller requests a spin lock that is already held, it typically spins in a loop testing whether the lock has become available. Such spinning wastes processor cycles so the lock should only be held for short durations and not across sleep / block operations. Callers should unlock spin locks before calling sleep operations.

  Spin locks are available on a variety of systems. The functions included in IEEE Std. 1003.1-200x are an attempt to standardize that existing practice.

- Lack of Timeout Feature

  Alternate versions of most blocking routines have been provided to support watchdog timeouts. No alternate interface of this sort has been provided for spin locks for the following reasons:

  - It is impossible to determine appropriate timeout intervals for spin locks in a portable manner. The amount of time one can expect to spend spin-waiting is inversely proportional to the degree of parallelism provided by the system.

    It can vary from a few cycles when each competing thread is running on its own processor, to an indefinite amount of time when all threads are multiplexed on a single processor (which is why spin locking is not advisable on uniprocessors).

  - When used properly, the amount of time the calling thread spends waiting on a spin lock should be considerably less than the time required to set up a corresponding watchdog timer. Since the primary purpose of spin locks is to provide a low-overhead synchronization mechanism for multi-processors, the overhead of a timeout mechanism was deemed unacceptable.

  It was also suggested that an additional *count* argument be provided (on the *pthread_spin_lock*( ) call) in *lieu* of a true timeout so that a spin lock call could fail gracefully if it was unable to apply the lock after *count* attempts. This idea was rejected because it is not existing practice. Furthermore, the same effect can be obtained with *pthread_spin_trylock*( ), as illustrated below:

```
int n = MAX_SPIN;

while ( --n >= 0 )
{
    if ( !pthread_spin_try_lock(...) )
        break;
}
if ( n >= 0 )
{
    /* Successfully acquired the lock */
}
else
{
    /* Unable to acquire the lock */
}
```

- *process-shared* Attribute

  The initialization functions associated with most POSIX synchronization objects (for example, mutexes, barriers, and read-write locks) take an attributes object with a *process-shared* attribute that specifies whether or not the object is to be shared across processes. In the draft corresponding to the first balloting round, two separate initialization functions are provided for spin locks, however: one for spin locks that were to be shared across processes (*spin_init*( )), and one for locks that were only used by multiple threads within a single process (*pthread_spin_init*( )). This was done so as to keep the overhead associated with spin waiting to an absolute minimum. However, the balloting group requested that, since the overhead associated to a bit check was small, spin locks should be consistent with the rest of the synchronization primitives, and thus the *process-shared* attribute was introduced for spin locks.

- Spin Locks *versus* Mutexes

  It has been suggested that mutexes are an adequate synchronization mechanism and spin locks are not necessary. Locking mechanisms typically must trade off the processor resources consumed while setting up to block the thread and the processor resources consumed by the thread while it is blocked. Spin locks require very little resources to set up the blocking of a thread. Existing practice is to simply loop, repeating the atomic locking operation until the lock is available. While the resources consumed to set up blocking of the thread are low, the thread continues to consume processor resources while it is waiting.

  On the other hand, mutexes may be implemented such that the processor resources consumed to block the thread are large relative to a spin lock. After detecting that the mutex lock is not available, the thread must alter its scheduling state, add itself to a set of waiting threads, and, when the lock becomes available again, undo all of this before taking over ownership of the mutex. However, while a thread is blocked by a mutex, no processor resources are consumed.

  Therefore, spin locks and mutexes may be implemented to have different characteristics. Spin locks may have lower overall overhead for very short-term blocking, and mutexes may have lower overall overhead when a thread will be blocked for longer periods of time. The presence of both interfaces allows implementations with these two different characteristics, both of which may be useful to a particular application.

  It has also been suggested that applications can build their own spin locks from the *pthread_mutex_trylock*( ) function:

```
while (pthread_mutex_trylock(&mutex));
```

The apparent simplicity of this construct is somewhat deceiving, however. While the actual wait is quite efficient, various guarantees on the integrity of mutex objects (for example, priority inheritance rules) may add overhead to the successful path of the trylock operation that is not required of spin locks. One could, of course, add an attribute to the mutex to bypass such overhead, but the very act of finding and testing this attribute represents more overhead than is found in the typical spin lock.

The need to hold spin lock overhead to an absolute minimum also makes it impossible to provide guarantees against starvation similar to those provided for mutexes or read-write locks. The overhead required to implement such guarantees (for example, disabling preemption before spinning) may well exceed the overhead of the spin wait itself by many orders of magnitude. If a ‘‘safe’’ spin wait seems desirable, it can always be provided (albeit at some performance cost) via appropriate mutex attributes.

XSI Supported Functions

On XSI-conformant systems, the following symbolic constants are always defined:

_POSIX_READER_WRITER_LOCKS
_POSIX_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_PROCESS_SHARED
_POSIX_THREADS

Therefore, the following threads functions are always supported:

pthread_atfork( )
pthread_attr_destroy( )
pthread_attr_getdetachstate( )
pthread_attr_getguardsize( )
pthread_attr_getschedparam( )
pthread_attr_getstackaddr( )
pthread_attr_getstacksize( )
pthread_attr_init( )
pthread_attr_setdetachstate( )
pthread_attr_setguardsize( )
pthread_attr_setschedparam( )
pthread_attr_setstackaddr( )
pthread_attr_setstacksize( )
pthread_cancel( )
pthread_cleanup_pop( )
pthread_cleanup_push( )
pthread_cond_broadcast( )
pthread_cond_destroy( )
pthread_cond_init( )
pthread_cond_signal( )
pthread_cond_timedwait( )
pthread_cond_wait( )
pthread_condattr_destroy( )
pthread_condattr_getpshared( )
pthread_condattr_init( )
pthread_key_delete( )
pthread_kill( )
pthread_mutex_destroy( )
pthread_mutex_init( )
pthread_mutex_lock( )
pthread_mutex_trylock( )
pthread_mutex_unlock( )
pthread_mutexattr_destroy( )
pthread_mutexattr_getpshared( )
pthread_mutexattr_gettype( )
pthread_mutexattr_init( )
pthread_mutexattr_setpshared( )
pthread_mutexattr_settype( )
pthread_once( )
pthread_rwlock_destroy( )
pthread_rwlock_init( )
pthread_rwlock_rdlock( )
pthread_rwlock_tryrdlock( )
pthread_rwlock_trywrlock( )
pthread_rwlock_unlock( )
pthread_rwlock_wrlock( )
pthread_rwlockattr_destroy( )
pthread_rwlockattr_getpshared( )
pthread_rwlockattr_init( )
pthread_rwlockattr_setpshared( )

pthread_condattr_setpshared( )
pthread_create( )
pthread_detach( )
pthread_equal( )
pthread_exit( )
pthread_getconcurrency( )
pthread_getspecific( )
pthread_join( )
pthread_key_create( )
pthread_self( )
pthread_setcancelstate( )
pthread_setcanceltype( )
pthread_setconcurrency( )
pthread_setspecific( )
pthread_sigmask( )
pthread_testcancel( )
sigwait( )

On XSI-conformant systems, the symbolic constant _POSIX_THREAD_SAFE_FUNCTIONS is always defined. Therefore, the following functions are always supported:

asctime_r( )
ctime_r( )
flockfile( )
ftrylockfile( )
funlockfile( )
getc_unlocked( )
getchar_unlocked( )
getgrgid_r( )
getgrnam_r( )
getpwnam_r( )
getpwuid_r( )
gmtime_r( )
localtime_r( )
putc_unlocked( )
putchar_unlocked( )
rand_r( )
readdir_r( )
strtok_r( )

The following threads functions are only supported on XSI-conformant systems if the Realtime Threads Option Group is supported :

pthread_attr_getinheritsched( )
pthread_attr_getschedpolicy( )
pthread_attr_getscope( )
pthread_attr_setinheritsched( )
pthread_attr_setschedpolicy( )
pthread_attr_setscope( )
pthread_getschedparam( )
pthread_mutex_getprioceiling( )
pthread_mutex_setprioceiling( )
pthread_mutexattr_getprioceiling( )
pthread_mutexattr_getprotocol( )
pthread_mutexattr_setprioceiling( )
pthread_mutexattr_setprotocol( )
pthread_setschedparam( )

XSI Threads Extensions

The following XSI extensions to POSIX.1c are now supported in IEEE Std. 1003.1-200x as part of the alignment with the Single UNIX Specification:

- Extended mutex attribute types
- Read-write locks and attributes (also introduced by IEEE Std. 1003.1j-2000 amendment)
- Thread concurrency level
- Thread stack guard size
- Parallel I / O

A total of 19 new functions were added.

These extensions carefully follow the threads programming model specified in POSIX.1c. As with POSIX.1c, all the new functions return zero if successful; otherwise, an error number is returned to indicate the error.

The concept of attribute objects was introduced in POSIX.1c to allow implementations to extend IEEE Std. 1003.1-200x without changing the existing interfaces. Attribute objects were defined for threads, mutexes, and condition variables. Attributes objects are defined as implementation-defined opaque types to aid extensibility, and functions are defined to allow attributes to be set or retrieved. This model has been followed when adding the new type attribute of **pthread_mutexattr_t** or the new read-write lock attributes object **pthread_rwlockattr_t**.

- Extended Mutex Attributes

  POSIX.1c defines a mutex attributes object as an implementation-defined opaque object of type **pthread_mutexattr_t**, and specifies a number of attributes which this object must have and a number of functions which manipulate these attributes. These attributes include *detachstate*, *inheritsched*, *schedparm*, *schedpolicy*, *contentionscope*, *stackaddr*, and *stacksize*.

  The System Interfaces volume of IEEE Std. 1003.1-200x specifies another mutex attribute called *type*. The *type* attribute allows applications to specify the behavior of mutex locking operations in situations where the POSIX.1c behavior is undefined. The OSF DCE threads implementation, based on Draft 4 of POSIX.1c, specified a similar attribute. Note that the names of the attributes have changed somewhat from the OSF DCE threads implementation.

  The System Interfaces volume of IEEE Std. 1003.1-200x also extends the specification of the following POSIX.1c functions which manipulate mutexes:

  pthread_mutex_lock( )
  pthread_mutex_trylock( )
  pthread_mutex_unlock( )

  to take account of the new mutex attribute type and to specify behavior which was declared as undefined in POSIX.1c. How a calling thread acquires or releases a mutex now depends upon the mutex *type* attribute.

  The *type* attribute can have the following values:

  PTHREAD_MUTEX_NORMAL
  Basic mutex with no specific error checking built in. Does not report a deadlock error.

  PTHREAD_MUTEX_RECURSIVE
  Allows any thread to recursively lock a mutex. The mutex must be unlocked an equal number of times to release the mutex.

  PTHREAD_MUTEX_ERRORCHECK
  Detects and reports simple usage errors; that is, an attempt to unlock a mutex that is not locked by the calling thread or that is not locked at all, or an attempt to relock a mutex the thread already owns.

  PTHREAD_MUTEX_DEFAULT
  The default mutex type. May be mapped to any of the above mutex types or may be an implementation-defined type.

  Normal mutexes do not detect deadlock conditions; for example, a thread will hang if it tries to relock a normal mutex that it already owns. Attempting to unlock a mutex locked by another thread, or unlocking an unlocked mutex, results in undefined behavior. Normal mutexes will usually be the fastest type of mutex available on a platform but provide the least error checking.

  Recursive mutexes are useful for converting old code where it is difficult to establish clear boundaries of synchronization. A thread can relock a recursive mutex without first unlocking it. The relocking deadlock which can occur with normal mutexes cannot occur with this type of mutex. However, multiple locks of a recursive mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. Furthermore, this type of mutex maintains the concept of an owner. Thus, a thread attempting to unlock a recursive mutex which another thread has locked returns with an error. A thread attempting to unlock a recursive mutex that is not locked shall return with an error. Never use a recursive mutex with condition variables because the implicit unlock performed by *pthread_cond_wait*( ) or *pthread_cond_timedwait*( ) will not actually release the mutex if it had been locked multiple times.

Errorcheck mutexes provide error checking and are useful primarily as a debugging aid. A thread attempting to relock an errorcheck mutex without first unlocking it returns with an error. Again, this type of mutex maintains the concept of an owner. Thus, a thread attempting to unlock an errorcheck mutex which another thread has locked returns with an error. A thread attempting to unlock an errorcheck mutex that is not locked also returns with an error. It should be noted that errorcheck mutexes will almost always be much slower than normal mutexes due to the extra state checks performed.

The *default* mutex type provides implementation-defined error checking. The default mutex may be mapped to one of the other defined types or may be something entirely different. This enables each vendor to provide the mutex semantics which the vendor feels will be most useful to their target users. Most vendors will probably choose to make normal mutexes the default so as to give applications the benefit of the fastest type of mutexes available on their platform. Check your implementation's documentation.

An application developer can use any of the mutex types almost interchangeably as long as the application does not depend upon the implementation detecting (or failing to detect) any particular errors. Note that a recursive mutex can be used with condition variable waits as long as the application never recursively locks the mutex.

Two functions are provided for manipulating the *type* attribute of a mutex attributes object. This attribute is set or returned in the *type* parameter of these functions. The *pthread_mutexattr_settype*( ) function is used to set a specific type value while *pthread_mutexattr_gettype*( ) is used to return the type of the mutex. Setting the *type* attribute of a mutex attributes object affects only mutexes initialized using that mutex attributes object. Changing the *type* attribute does not affect mutexes previously initialized using that mutex attributes object.

- Read-Write Locks and Attributes

The read-write locks introduced have been harmonized with those in IEEE Std. 1003.1j-2000; see also Section B.2.9.6 (on page 3472).

Read-write locks (also known as reader-writer locks) allow a thread to exclusively lock some shared data while updating that data, or allow any number of threads to have simultaneous read-only access to the data.

Unlike a mutex, a read-write lock distinguishes between reading data and writing data. A mutex excludes all other threads. A read-write lock allows other threads access to the data, providing no thread is modifying the data. Thus, a read-write lock is less primitive than either a mutex-condition variable pair or a semaphore.

Application developers should consider using a read-write lock rather than a mutex to protect data that is frequently referenced but seldom modified. Most threads (readers) will be able to read the data without waiting and will only have to block when some other thread (a writer) is in the process of modifying the data. Conversely a thread that wants to change the data is forced to wait until there are no readers. This type of lock is often used to facilitate parallel access to data on multi-processor platforms or to avoid context switches on single processor platforms where multiple threads access the same data.

If a read-write lock becomes unlocked and there are multiple threads waiting to acquire the write lock, the implementation's scheduling policy determines which thread shall acquire the read-write lock for writing. If there are multiple threads blocked on a read-write lock for both read locks and write locks, it is unspecified whether the readers or a writer acquire the lock first. However, for performance reasons, implementations often favor writers over readers to avoid potential writer starvation.

A read-write lock object is an implementation-defined opaque object of type **pthread_rwlock_t** as defined in **<pthread.h>**. There are two different sorts of locks associated with a read-write lock: a *read lock* and a *write lock*.

The *pthread_rwlockattr_init*( ) function initializes a read-write lock attributes object with the default value for all the attributes defined in the implementation. After a read-write lock attributes object has been used to initialize one or more read-write locks, changes to the read-write lock attributes object, including destruction, do not affect previously initialized read-write locks.

Implementations must provide at least the read-write lock attribute *process-shared*. This attribute can have the following values:

PTHREAD_PROCESS_SHARED
: Any thread of any process that has access to the memory where the read-write lock resides can manipulate the read-write lock.

PTHREAD_PROCESS_PRIVATE
: Only threads created within the same process as the thread that initialized the read-write lock can manipulate the read-write lock. This is the default value.

The *pthread_rwlockattr_setpshared*( ) function is used to set the *process-shared* attribute of an initialized read-write lock attributes object while the function *pthread_rwlockattr_getpshared*( ) obtains the current value of the *process-shared* attribute.

A read-write lock attributes object is destroyed using the *pthread_rwlockattr_destroy*( ) function. The effect of subsequent use of the read-write lock attributes object is undefined.

A thread creates a read-write lock using the *pthread_rwlock_init*( ) function. The attributes of the read-write lock can be specified by the application developer; otherwise, the default implementation-defined read-write lock attributes are used if the pointer to the read-write lock attributes object is NULL. In cases where the default attributes are appropriate, the PTHREAD_RWLOCK_INITIALIZER macro can be used to initialize statically allocated read-write locks.

A thread which wants to apply a read lock to the read-write lock can use either *pthread_rwlock_rdlock*( ) or *pthread_rwlock_tryrdlock*( ). If *pthread_rwlock_rdlock*( ) is used, the thread acquires a read lock if a writer does not hold the write lock and there are no writers blocked on the write lock. If a read lock is not acquired, the calling thread blocks until it can acquire a lock. However, if *pthread_rwlock_tryrdlock*( ) is used, the function returns immediately with the error [EBUSY] if any thread holds a write lock or there are blocked writers waiting for the write lock.

A thread which wants to apply a write lock to the read-write lock can use either of two functions: *pthread_rwlock_wrlock*( ) or *pthread_rwlock_trywrlock*( ). If *pthread_rwlock_wrlock*( ) is used, the thread acquires the write lock if no other reader or writer threads hold the read-write lock. If the write lock is not acquired, the thread blocks until it can acquire the write lock. However, if *pthread_rwlock_trywrlock*( ) is used, the function returns immediately with the error [EBUSY] if any thread is holding either a read or a write lock.

The *pthread_rwlock_unlock*( ) function is used to unlock a read-write lock object held by the calling thread. Results are undefined if the read-write lock is not held by the calling thread. If there are other read locks currently held on the read-write lock object, the read-write lock object shall remain in the read locked state but without the current thread as one of its owners. If this function releases the last read lock for this read-write lock object, the read-write lock object shall be put in the unlocked read state. If this function is called to release a write lock for this read-write lock object, the read-write lock object shall be put in the unlocked state.

- Thread Concurrency Level

  On threads implementations that multiplex user threads onto a smaller set of kernel execution entities, the system attempts to create a reasonable number of kernel execution entities for the application upon application startup.

  On some implementations, these kernel entities are retained by user threads that block in the kernel. Other implementations do not *timeslice* user threads so that multiple compute-bound user threads can share a kernel thread. On such implementations, some applications may use up all the available kernel execution entities before its user-space threads are used up. The process may be left with user threads capable of doing work for the application but with no way to schedule them.

  The *pthread_setconcurrency*( ) function enables an application to request more kernel entities; that is, specify a desired concurrency level. However, this function merely provides a hint to the implementation. The implementation is free to ignore this request or to provide some other number of kernel entities. If an implementation does not multiplex user threads onto a smaller number of kernel execution entities, the *pthread_setconcurrency*( ) function has no effect.

  The *pthread_setconcurrency*( ) function may also have an effect on implementations where the kernel mode and user mode schedulers cooperate to ensure that ready user threads are not prevented from running by other threads blocked in the kernel.

  The *pthread_getconcurrency*( ) function always returns the value set by a previous call to *pthread_setconcurrency*( ). However, if *pthread_setconcurrency*( ) was not previously called, this function shall return zero to indicate that the threads implementation is maintaining the concurrency level.

- Thread Stack Guard Size

  DCE threads introduced the concept of a *thread stack guard size*. Most thread implementations add a region of protected memory to a thread's stack, commonly known as a *guard region*, as a safety measure to prevent stack pointer overflow in one thread from corrupting the contents of another thread's stack. The default size of the guard regions attribute is {PAGESIZE} bytes and is implementation-defined.

  Some application developers may wish to change the stack guard size. When an application creates a large number of threads, the extra page allocated for each stack may strain system resources. In addition to the extra page of memory, the kernel's memory manager has to keep track of the different protections on adjoining pages. When this is a problem, the application developer may request a guard size of 0 bytes to conserve system resources by eliminating stack overflow protection.

  Conversely an application that allocates large data structures such as arrays on the stack may wish to increase the default guard size in order to detect stack overflow. If a thread allocates two pages for a data array, a single guard page provides little protection against thread stack overflows since the thread can corrupt adjoining memory beyond the guard page.

The System Interfaces volume of IEEE Std. 1003.1-200x defines a new attribute of a thread attributes object; that is, the *guardsize* attribute which allows applications to specify the size of the guard region of a thread's stack.

Two functions are provided for manipulating a thread's stack guard size. The *pthread_attr_setguardsize*( ) function sets the thread *guardsize* attribute, and the *pthread_attr_getguardsize*( ) function retrieves the current value.

An implementation may round up the requested guard size to a multiple of the configurable system variable {PAGESIZE}. In this case, *pthread_attr_getguardsize*( ) returns the guard size specified by the previous *pthread_attr_setguardsize*( ) function call and not the rounded up value.

If an application is managing its own thread stacks using the *stackaddr* attribute, the *guardsize* attribute is ignored and no stack overflow protection is provided. In this case, it is the responsibility of the application to manage stack overflow along with stack allocation.

- Parallel I/O

  Many I/O intensive applications, such as database engines, attempt to improve performance through the use of parallel I/O. However, POSIX.1 does not support parallel I/O very well because the current offset of a file is an attribute of the file descriptor.

  Suppose two or more threads independently issue read requests on the same file. To read specific data from a file, a thread must first call *lseek*( ) to seek to the proper offset in the file, and then call *read*( ) to retrieve the required data. If more than one thread does this at the same time, the first thread may complete its seek call, but before it gets a chance to issue its read call a second thread may complete its seek call, resulting in the first thread accessing incorrect data when it issues its read call. One workaround is to lock the file descriptor while seeking and reading or writing, but this reduces parallelism and adds overhead.

  Instead, the System Interfaces volume of IEEE Std. 1003.1-200x provides two functions to make seek/read and seek/write operations atomic. The file descriptor's current offset is unchanged, thus allowing multiple read and write operations to proceed in parallel. This improves the I/O performance of threaded applications. The *pread*( ) function is used to do an atomic read of data from a file into a buffer. Conversely, the *pwrite*( ) function does an atomic write of data from a buffer to a file.

B.2.9.1 Thread-Safety

All functions required by IEEE Std. 1003.1-200x need to be thread-safe. Implementations have to provide internal synchronization when necessary in order to achieve this goal. In certain cases—for example, most floating-point implementations—context switch code may have to manage the writable shared state.

It is not required that all functions provided by IEEE Std. 1003.1-200x be either async-cancel-safe or async-signal-safe.

As it turns out, some functions are inherently not thread-safe; that is, their interface specifications preclude reentrancy. For example, some functions (such as *asctime*( )) return a pointer to a result stored in memory space allocated by the function on a per-process basis. Such a function is not thread-safe, because its result can be overwritten by successive invocations. Other functions, while not inherently non-thread-safe, may be implemented in ways that lead to them not being thread-safe. For example, some functions (such as *rand*( )) store state information (such as a seed value, which survives multiple function invocations) in memory space allocated by the function on a per-process basis. The implementation of such a function is not thread-safe if the implementation fails to synchronize invocations of the function and thus fails to protect the state information. The problem is that when the state information is not protected, concurrent invocations can interfere with one another (for example, see the same seed value).

Thread-Safety and Locking of Existing Functions

Originally, POSIX.1 was not designed to work in a multi-threaded environment, and some implementations of some existing functions will not work properly when executed concurrently. To provide routines that will work correctly in an environment with threads (‘‘thread-safe’’), two problems need to be solved:

1. Routines that maintain or return pointers to static areas internal to the routine (which may now be shared) need to be modified. The routines *ttyname*( ) and *localtime*( ) are examples.

2. Routines that access data space shared by more than one thread need to be modified. The *malloc*( ) function and the *stdio* family routines are examples.

There are a variety of constraints on these changes. The first is compatibility with the existing versions of these functions—non-thread-safe functions will continue to be in use for some time, as the original interfaces are used by existing code. Another is that the new thread-safe versions of these functions represent as small a change as possible over the familiar interfaces provided by the existing non-thread-safe versions. The new interfaces should be independent of any particular threads implementation. In particular, they should be thread-safe without depending on explicit thread-specific memory. Finally, there should be minimal performance penalty due to the changes made to the functions.

It is intended that the list of functions from POSIX.1 that cannot be made thread-safe and for which corrected versions are provided be complete.

Thread-Safety and Locking Solutions

Many of the POSIX.1 functions were thread-safe and did not change at all. However, some functions (for example, the math functions typically found in **libm**) are not thread-safe because of writable shared global state. For instance, in IEEE Std. 754-1985 floating-point implementations, the computation modes and flags are global and shared.

Some functions are not thread-safe because a particular implementation is not reentrant, typically because of a non-essential use of static storage. These require only a new implementation.

Thread-safe libraries are useful in a wide range of parallel (and asynchronous) programming environments, not just within pthreads. In order to be used outside the context of pthreads, however, such libraries still have to use some synchronization method. These could either be independent of the pthread synchronization operations, or they could be a subset of the pthread interfaces. Either method results in thread-safe library implementations that can be used without the rest of pthreads.

Some functions, such as the *stdio* family interface and dynamic memory allocation functions such as *malloc*( ), are interdependent routines that share resources (for example, buffers) across related calls. These require synchronization to work correctly, but they do not require any change to their external (user-visible) interfaces.

In some cases, such as *getc*( ) and *putc*( ), adding synchronization is likely to create an unacceptable performance impact. In this case, slower thread-safe synchronized functions are to be provided, but the original, faster (but unsafe) functions (which may be implemented as macros) are retained under new names. Some additional special-purpose synchronization facilities are necessary for these macros to be usable in multi-threaded programs. This also requires changes in **<stdio.h>**.

The other common reason that functions are unsafe is that they return a pointer to static storage, making the functions non-thread-safe. This has to be changed, and there are three natural choices:

1. Return a pointer to thread-specific storage

   This could incur a severe performance penalty on those architectures with a costly implementation of the thread-specific data interface.

   A variation on this technique is to use *malloc*( ) to allocate storage for the function output and return a pointer to this storage. This technique may also have an undesirable performance impact, however, and a simplistic implementation requires that the user program explicitly free the storage object when it is no longer needed. This technique is used by some existing POSIX.1 functions. With careful implementation for infrequently used functions, there may be little or no performance or storage penalty, and the maintenance of already-standardized interfaces is a significant benefit.

2. Return the actual value computed by the function

   This technique can only be used with functions that return pointers to structures—routines that return character strings would have to wrap their output in an enclosing structure in order to return the output on the stack. There is also a negative performance impact inherent in this solution in that the output value has to be copied twice before it can be used by the calling function: once from the called routine's local buffers to the top of the stack, then from the top of the stack to the assignment target. Finally, many older compilers cannot support this technique due to a historical tendency to use internal static buffers to deliver the results of structure-valued functions.

3. Have the caller pass the address of a buffer to contain the computed value

   The only disadvantage of this approach is that extra arguments have to be provided by the calling program. It represents the most efficient solution to the problem, however, and, unlike the *malloc*( ) technique, it is semantically clear.

There are some routines (often groups of related routines) whose interfaces are inherently non-thread-safe because they communicate across multiple function invocations by means of static memory locations. The solution is to redesign the calls so that they are thread-safe, typically by passing the needed data as extra parameters. Unfortunately, this may require major changes to the interface as well.

A floating-point implementation using IEEE Std. 754-1985 is a case in point. A less problematic example is the *rand48* family of pseudo-random number generators. The functions *getgrgid*( ), *getgrnam*( ), *getpwnam*( ), and *getpwuid*( ) are another such case.

The problems with *errno* are discussed in **Alternative Solutions for Per-Thread errno** (on page 3390).

Some functions can be thread-safe or not, depending on their arguments. These include the *tmpnam*( ) and *ctermid*( ) functions. These functions have pointers to character strings as arguments. If the pointers are not NULL, the functions store their results in the character string; however, if the pointers are NULL, the functions store their results in an area that may be static and thus subject to overwriting by successive calls. These should only be called by multi-thread applications when their arguments are non-NULL.

Asynchronous Safety and Thread-Safety

A floating-point implementation has many modes that effect rounding and other aspects of computation. Functions in some math library implementations may change the computation modes for the duration of a function call. If such a function call is interrupted by a signal or cancelation, the floating-point state is not required to be protected.

There is a significant cost to make floating-point operations async-cancel-safe or async-signal-safe; accordingly, neither form of async safety is required.

Functions Returning Pointers to Static Storage

For those functions that are not thread-safe because they return values in fixed size statically allocated structures, alternate ''_r'' forms are provided that pass a pointer to an explicit result structure. Those that return pointers into library-allocated buffers have forms provided with explicit buffer and length parameters.

For functions that return pointers to library-allocated buffers, it makes sense to provide ''_r'' versions that allow the application control over allocation of the storage in which results are returned. This allows the state used by these functions to be managed on an application-specific basis, supporting per-thread, per-process, or other application-specific sharing relationships.

Early proposals had provided ''_r'' versions for functions that returned pointers to variable-size buffers without providing a means for determining the required buffer size. This would have made using such functions exceedingly clumsy, potentially requiring iteratively calling them with increasingly larger guesses for the amount of storage required. Hence, *sysconf*( ) variables have been provided for such functions that return the maximum required buffer size.

Thus, the rule that has been followed by IEEE Std. 1003.1-200x when adapting single-threaded non-thread-safe library functions is as follows: all functions returning pointers to library-allocated storage should have ''_r'' versions provided, allowing the application control over the storage allocation. Those with variable-sized return values accept both a buffer address and a length parameter. The *sysconf*( ) variables are provided to supply the appropriate buffer sizes when required. Implementors are encouraged to apply the same rule when adapting their own existing functions to a pthreads environment.

B.2.9.2 Thread IDs

Separate programs should communicate through well-defined interfaces and should not depend on each other's implementation. For example, if a programmer decides to rewrite the *sort* program using multiple threads, it should be easy to do this so that the interface to the *sort* program does not change. Consider that if the user causes SIGINT to be generated while the *sort* program is running, keeping the same interface means that the entire sort program is killed, not just one of its threads. As another example, consider a realtime program that manages a reactor. Such a program may wish to allow other programs to control the priority at which it watches the control rods. One technique to accomplish this is to write the ID of the thread watching the control rods into a file and allow other programs to change the priority of that thread as they see fit. A simpler technique is to have the reactor process accept IPCs (Inter-Process Communication messages) from other processes, telling it at a semantic level what priority the program should assign to watching the control rods. This allows the programmer greater flexibility in the implementation. For example, the programmer can change the implementation from having one thread per rod to having one thread watching all of the rods without changing the interface. Having threads live inside the process means that the implementation of a process is invisible to outside processes (excepting debuggers and system management tools).

Threads do not provide a protection boundary. Every thread model allows threads to share memory with other threads and encourages this sharing to be widespread. This means that one thread can wipe out memory that is needed for the correct functioning of other threads that are sharing its memory. Consequently, providing each thread with its own user and/or group IDs would not provide a protection boundary between threads sharing memory.

B.2.9.3 Thread Mutexes

There is no additional rationale for this section.

B.2.9.4 Thread Scheduling

- Scheduling Implementation Models

  The following scheduling implementation models are presented in terms of threads and ''kernel entities''. This is to simplify exposition of the models, and it does not imply that an implementation actually has an identifiable ''kernel entity''.

  A kernel entity is not defined beyond the fact that it has scheduling attributes that are used to resolve contention with other kernel entities for execution resources. A kernel entity may be thought of as an envelope that holds a thread or a separate kernel thread. It is not a conventional process, although it shares with the process the attribute that it has a single thread of control; it does not necessarily imply an address space, open files, and so on. It is better thought of as a primitive facility upon which conventional processes and threads may be constructed.

  — System Thread Scheduling Model

    This model consists of one thread per kernel entity. The kernel entity is solely responsible for scheduling thread execution on one or more processors. This model schedules all threads against all other threads in the system using the scheduling attributes of the thread.

  — Process Scheduling Model

    A generalized process scheduling model consists of two levels of scheduling. A threads library creates a pool of kernel entities, as required, and schedules threads to run on them using the scheduling attributes of the threads. Typically, the size of the pool is a function of the simultaneously runnable threads, not the total number of threads. The kernel then schedules the kernel entities onto processors according to their scheduling attributes, which are managed by the threads library. This set model potentially allows a wide range of mappings between threads and kernel entities.

- System and Process Scheduling Model Performance

  There are a number of important implications on the performance of applications using these scheduling models. The process scheduling model potentially provides lower overhead for making scheduling decisions, since there is no need to access kernel-level information or functions and the set of schedulable entities is smaller (only the threads within the process).

  On the other hand, since the kernel is also making scheduling decisions regarding the system resources under its control (for example, CPU(s), I/O devices, memory), decisions that do not take thread scheduling parameters into account can result in indeterminate delays for realtime application threads, causing them to miss maximum response time limits.

- Rate Monotonic Scheduling

  Rate monotonic scheduling was considered, but rejected for standardization in the context of pthreads. A sporadic server policy is included.

- Scheduling Options

  In IEEE Std. 1003.1-200x, the basic thread scheduling functions are defined under the Threads option, so that they are required of all threads implementations. However, there are no specific scheduling policies required by this option to allow for conforming thread implementations that are not targeted to realtime applications.

Specific standard scheduling policies are defined to be under the Thread Execution Scheduling option, and they are specifically designed to support realtime applications by providing predictable resource sharing sequences. The name of this option was chosen to emphasize that this functionality is defined as appropriate for realtime applications that require simple priority-based scheduling.

It is recognized that these policies are not necessarily satisfactory for some multi-processor implementations, and work is ongoing to address a wider range of scheduling behaviors. The interfaces have been chosen to create abundant opportunity for future scheduling policies to be implemented and standardized based on this interface. In order to standardize a new scheduling policy, all that is required (from the standpoint of thread scheduling attributes) is to define a new policy name, new members of the thread attributes object, and functions to set these members when the scheduling policy is equal to the new value.

Scheduling Contention Scope

In order to accommodate the requirement for realtime response, each thread has a scheduling contention scope attribute. Threads with a system scheduling contention scope have to be scheduled with respect to all other threads in the system. These threads are usually bound to a single kernel entity that reflects their scheduling attributes and are directly scheduled by the kernel.

Threads with a process scheduling contention scope need be scheduled only with respect to the other threads in the process. These threads may be scheduled within the process onto a pool of kernel entities. The implementation is also free to bind these threads directly to kernel entities and let them be scheduled by the kernel. Process scheduling contention scope allows the implementation the most flexibility and is the default if both contention scopes are supported and none is specified.

Thus, the choice by implementors to provide one or the other (or both) of these scheduling models is driven by the need of their supported application domains for worst-case (that is, realtime) response, or average-case (non-realtime) response.

Scheduling Allocation Domain

The SCHED_FIFO and SCHED_RR scheduling policies take on different characteristics on a multi-processor. Other scheduling policies are also subject to changed behavior when executed on a multi-processor. The concept of scheduling allocation domain determines the set of processors on which the threads of an application may run. By considering the application's processor scheduling allocation domain for its threads, scheduling policies can be defined in terms of their behavior for varying processor scheduling allocation domain values. It is conceivable that not all scheduling allocation domain sizes make sense for all scheduling policies on all implementations. The concept of scheduling allocation domain, however, is a useful tool for the description of multi-processor scheduling policies.

The ''process control'' approach to scheduling obtains significant performance advantages from dynamic scheduling allocation domain sizes when it is applicable.

Non-Uniform Memory Access (NUMA) multi-processors may use a system scheduling structure that involves reassignment of threads among scheduling allocation domains. In NUMA machines, a natural model of scheduling is to match scheduling allocation domains to clusters of processors. Load balancing in such an environment requires changing the scheduling allocation domain to which a thread is assigned.

Scheduling Documentation

Implementation-provided scheduling policies need to be completely documented in order to be useful. This documentation includes a description of the attributes required for the policy, the scheduling interaction of threads running under this policy and all other supported policies, and the effects of all possible values for processor scheduling allocation domain. Note that for the implementor wishing to be minimally-compliant, it is (minimally) acceptable to define the behavior as undefined.

Scheduling Contention Scope Attribute

The scheduling contention scope defines how threads compete for resources. Within IEEE Std. 1003.1-200x, scheduling contention scope is used to describe only how threads are scheduled in relation to one another in the system. That is, either they are scheduled against all other threads in the system (''system scope'') or only against those threads in the process (''process scope''). In fact, scheduling contention scope may apply to additional resources, including virtual timers and profiling, which are not currently considered by IEEE Std. 1003.1-200x.

Mixed Scopes

If only one scheduling contention scope is supported, the scheduling decision is straightforward. To perform the processor scheduling decision in a mixed scope environment, it is necessary to map the scheduling attributes of the thread with process-wide contention scope to the same attribute space as the thread with system-wide contention scope.

Since a conforming implementation has to support one and may support both scopes, it is useful to discuss the effects of such choices with respect to example applications. If an implementation supports both scopes, mixing scopes provides a means of better managing system-level (that is, kernel-level) and library-level resources. In general, threads with system scope will require the resources of a separate kernel entity in order to guarantee the scheduling semantics. On the other hand, threads with process scope can share the resources of a kernel entity while maintaining the scheduling semantics.

The application is free to create threads with dedicated kernel resources, and other threads that multiplex kernel resources. Consider the example of a window server. The server allocates two threads per widget: one thread manages the widget user interface (including drawing), while the other thread takes any required application action. This allows the widget to be ''active'' while the application is computing. A screen image may be built from thousands of widgets. If each of these threads had been created with system scope, then most of the kernel-level resources might be wasted, since only a few widgets are active at any one time. In addition, mixed scope is particularly useful in a window server where one thread with high priority and system scope handles the mouse so that it tracks well. As another example, consider a database server. For each of the hundreds or thousands of clients supported by a large server, an equivalent number of threads will have to be created. If each of these threads were system, the consequences would be the same as for the window server example above. However, the server could be constructed so that actual retrieval of data is done by several dedicated threads. Dedicated threads that do work for all clients frequently justify the added expense of system scope. If it were not permissible to mix system and process threads in the same process, this type of solution would not be possible.

Dynamic Thread Scheduling Parameters Access

In many time-constrained applications, there is no need to change the scheduling attributes dynamically during thread or process execution, since the general use of these attributes is to reflect directly the time constraints of the application. Since these time constraints are generally imposed to meet higher-level system requirements, such as accuracy or availability, they frequently should remain unchanged during application execution.

However, there are important situations in which the scheduling attributes should be changed. Generally, this will occur when external environmental conditions exist in which the time constraints change. Consider, for example, a space vehicle major mode change, such as the change from ascent to descent mode, or the change from the space environment to the atmospheric environment. In such cases, the frequency with which many of the sensors or acutators need to be read or written will change, which will necessitate a priority change. In other cases, even the existence of a time constraint might be temporary, necessitating not just a priority change, but also a policy change for ongoing threads or processes. For this reason, it is critical that the interface should provide functions to change the scheduling parameters dynamically, but, as with many of the other realtime functions, it is important that applications use them properly to avoid the possibility of unnecessarily degrading performance.

In providing functions for dynamically changing the scheduling behavior of threads, there were two options: provide functions to get and set the individual scheduling parameters of threads, or provide a single interface to get and set all the scheduling parameters for a given thread simultaneously. Both approaches have merit. Access functions for individual parameters allow simpler control of thread scheduling for simple thread scheduling parameters. However, a single function for setting all the parameters for a given scheduling policy is required when first setting that scheduling policy. Since the single all-encompassing functions are required, it was decided to leave the interface as minimal as possible. Note that simpler functions (such as *pthread_setprio*( ) for threads running under the priority-based schedulers) can be easily defined in terms of the all-encompassing functions.

If the *pthread_setschedparam*( ) function executes successfully, it will have set all of the scheduling parameter values indicated in *param*; otherwise, none of the scheduling parameters will have been modified. This is necessary to ensure that the scheduling of this and all other threads continues to be consistent in the presence of an erroneous scheduling parameter.

The [EPERM] error value is included in the list of possible *pthread_setschedparam*( ) error returns as a reflection of the fact that the ability to change scheduling parameters increases risks to the implementation and application performance if the scheduling parameters are changed improperly. For this reason, and based on some existing practice, it was felt that some implementations would probably choose to define specific permissions for changing either a thread's own or another thread's scheduling parameters. IEEE Std. 1003.1-200x does not include portable methods for setting or retrieving permissions, so any such use of permissions is completely unspecified .

Mutex Initialization Scheduling Attributes

In a priority-driven environment, a direct use of traditional primitives like mutexes and condition variables can lead to unbounded priority inversion, where a higher priority thread can be blocked by a lower priority thread, or set of threads, for an unbounded duration of time. As a result, it becomes impossible to guarantee thread deadlines. Priority inversion can be bounded and minimized by the use of priority inheritance protocols. This allows thread deadlines to be guaranteed even in the presence of synchronization requirements.

Two useful but simple members of the family of priority inheritance protocols are the basic priority inheritance protocol and the priority ceiling protocol emulation. Under the Basic Priority

Inheritance protocol (governed by the Threads Priority Inheritance option), a thread that is blocking higher priority threads executes at the priority of the highest priority thread that it blocks. This simple mechanism allows priority inversion to be bounded by the duration of critical sections and makes timing analysis possible.

Under the Priority Ceiling Protocol Emulation protocol (governed by the Thread Priority Protection option), each mutex has a priority ceiling, usually defined as the priority of the highest priority thread that can lock the mutex. When a thread is executing inside critical sections, its priority is unconditionally increased to the highest of the priority ceilings of all the mutexes owned by the thread. This protocol has two very desirable properties in uni-processor systems. First, a thread can be blocked by a lower priority thread for at most the duration of one single critical section. Furthermore, when the protocol is correctly used in a single processor, and if threads do not become blocked while owning mutexes, mutual deadlocks are prevented.

The priority ceiling emulation can be extended to multiple processor environments, in which case the values of the priority ceilings will be assigned depending on the kind of mutex that is being used: local to only one processor, or global, shared by several processors. Local priority ceilings will be assigned the usual way, equal to the priority of the highest priority thread that may lock that mutex. Global priority ceilings will usually be assigned a priority level higher than all the priorities assigned to any of the threads that reside in the involved processors to avoid the effect called remote blocking.

Change the Priority Ceiling of a Mutex

In order for the priority protect protocol to exhibit its desired properties of bounding priority inversion and avoidance of deadlock, it is critical that the ceiling priority of a mutex be the same as the priority of the highest thread that can ever hold it, or higher. Thus, if the priorities of the threads using such mutexes never change dynamically, there is no need ever to change the priority ceiling of a mutex.

However, if a major system mode change results in an altered response time requirement for one or more application threads, their priority has to change to reflect it. It will occasionally be the case that the priority ceilings of mutexes held also need to change. While changing priority ceilings should generally be avoided, it is important that IEEE Std. 1003.1-200x provide these interfaces for those cases in which it is necessary.

B.2.9.5 Thread Cancelation

Many existing threads packages have facilities for canceling an operation or canceling a thread. These facilities are used for implementing user requests (such as the CANCEL button in a window-based application), for implementing OR parallelism (for example, telling the other threads to stop working once one thread has found a forced mate in a parallel chess program), or for implementing the ABORT mechanism in Ada.

POSIX programs traditionally have used the signal mechanism combined with either *longjmp*( ) or polling to cancel operations. Many POSIX programmers have trouble using these facilities to solve their problems efficiently in a single-threaded process. With the introduction of threads, these solutions become even more difficult to use.

The main issues with implementing a cancelation facility are specifying the operation to be canceled, cleanly releasing any resources allocated to that operation, controlling when the target notices that it has been canceled, and defining the interaction between asynchronous signals and cancelation.

Specifying the Operation to Cancel

Consider a thread that calls through five distinct levels of program abstraction and then, inside the lowest-level abstraction, calls a function that suspends the thread. (An abstraction boundary is a layer at which the client of the abstraction sees only the service being provided and can remain ignorant of the implementation. Abstractions are often layered, each level of abstraction being a client of the lower-level abstraction and implementing a higher-level abstraction.) Depending on the semantics of each abstraction, one could imagine wanting to cancel only the call that causes suspension, only the bottom two levels, or the operation being done by the entire thread. Canceling operations at a finer grain than the entire thread is difficult because threads are active and they may be run in parallel on a multi-processor. By the time one thread can make a request to cancel an operation, the thread performing the operation may have completed that operation and gone on to start another operation whose cancelation is not desired. Thread IDs are not reused until the thread has exited, and either it was created with the *Attr detachstate* attribute set to PTHREAD_CREATE_DETACHED or the *pthread_join*( ) or *pthread_detach*( ) function has been called for that thread. Consequently, a thread cancelation will never be misdirected when the thread terminates. For these reasons, the canceling of operations is done at the granularity of the thread. Threads are designed to be inexpensive enough so that a separate thread may be created to perform each separately cancelable operation; for example, each possibly long running user request.

For cancelation to be used in existing code, cancelation scopes and handlers will have to be established for code that needs to release resources upon cancelation, so that it follows the programming discipline described in the text.

A Special Signal Versus a Special Interface

Two different mechanisms were considered for providing the cancelation interfaces. The first was to provide an interface to direct signals at a thread and then to define a special signal that had the required semantics. The other alternative was to use a special interface that delivered the correct semantics to the target thread.

The solution using signals produced a number of problems. It required the implementation to provide cancelation in terms of signals whereas a perfectly valid (and possibly more efficient) implementation could have both layered on a low-level set of primitives. There were so many exceptions to the special signal (it cannot be used with kill, no POSIX.1 interfaces can be used with it) that it was clearly not a valid signal. Its semantics on delivery were also completely different from any existing POSIX.1 signal. As such, a special interface that did not mandate the implementation and did not confuse the semantics of signals and cancelation was felt to be the better solution.

Races Between Cancelation and Resuming Execution

Due to the nature of cancelation, there is generally no synchronization between the thread requesting the cancelation of a blocked thread and events that may cause that thread to resume execution. For this reason, and because excess serialization hurts performance, when both an event that a thread is waiting for has occurred and a cancelation request has been made and cancelation is enabled, IEEE Std. 1003.1-200x explicitly allows the implementation to choose between returning from the blocking call or acting on the cancelation request.

Interaction of Cancelation with Asynchronous Signals

A typical use of cancelation is to acquire a lock on some resource and to establish a cancelation cleanup handler for releasing the resource when and if the thread is canceled.

A correct and complete implementation of cancelation in the presence of asynchronous signals requires considerable care. An implementation has to push a cancelation cleanup handler on the cancelation cleanup stack while maintaining the integrity of the stack data structure. If an asynchronously generated signal is posted to the thread during a stack operation, the signal handler cannot manipulate the cancelation cleanup stack. As a consequence, asynchronous signal handlers may not cancel threads or otherwise manipulate the cancelation state of a thread. Threads may, of course, be canceled by another thread that used a *sigwait*( ) function to wait synchronously for an asynchronous signal.

In order for cancelation to function correctly, it is required that asynchronous signal handlers not change the cancelation state. This requires that some elements of existing practice, such as using *longjmp*( ) to exit from an asynchronous signal handler implicitly, be prohibited in cases where the integrity of the cancelation state of the interrupt thread cannot be ensured.

Thread Cancelation Overview

- Cancelability States

  The three possible cancelability states (disabled, deferred, and asynchronous) are encoded into two separate bits ((disable, enable) and (deferred, asynchronous)) to allow them to be changed and restored independently. For instance, short code sequences that will not block sometimes disable cancelability on entry and restore the previous state upon exit. Likewise, long or unbounded code sequences containing no convenient explicit cancelation points will sometimes set the cancelability type to asynchronous on entry and restore the previous value upon exit.

- Cancelation Points

  Cancelation points are points inside of certain functions where a thread has to act on any pending cancelation request when cancelability is enabled, if the function would block. As with checking for signals, operations need only check for pending cancelation requests when the operation is about to block indefinitely.

  The idea was considered of allowing implementations to define whether blocking calls such as *read*( ) should be cancelation points. It was decided that it would adversely affect the design of portable applications if blocking calls were not cancelation points because threads could be left blocked in an uncancelable state.

  There are several important blocking routines that are specifically not made cancelation points:

  — *pthread_mutex_lock*( )

    If *pthread_mutex_lock*( ) were a cancelation point, every routine that called it would also become a cancelation point (that is, any routine that touched shared state would automatically become a cancelation point). For example, *malloc*( ), *free*( ), and *rand*( ) would become cancelation points under this scheme. Having too many cancelation points makes programming very difficult, leading to either much disabling and restoring of cancelability or much difficulty in trying to arrange for reliable cleanup at every possible place.

    Since *pthread_mutex_lock*( ) is not a cancelation point, threads could result in being blocked uninterruptibly for long periods of time if mutexes were used as a general synchronization mechanism. As this is normally not acceptable, mutexes should only be used to protect resources that are held for small fixed lengths of time where not being able to be canceled will not be a problem. Resources that need to be held exclusively for long periods of time should be protected with condition variables.

— *barrier_wait*( )

Canceling a barrier wait will render a barrier unusable. Similar to a barrier timeout (which the standard developers rejected), there is no way to guarantee the consistency of a barrier's internal data structures if a barrier wait is canceled.

— *pthread_spin_lock*( )

As with mutexes, spin locks should only be used to protect resources that are held for small fixed lengths of time where not being cancelable will not be a problem.

Every library routine should specify whether or not it includes any cancelation points. Typically, only those routines that may block or compute indefinitely need to include cancelation points.

Correctly coded routines only reach cancelation points after having set up a cancelation cleanup handler to restore invariants if the thread is canceled at that point. Being cancelable only at specified cancelation points allows programmers to keep track of actions needed in a cancelation cleanup handler more easily. A thread should only be made asynchronously cancelable when it is not in the process of acquiring or releasing resources or otherwise in a state from which it would be difficult or impossible to recover.

• Thread Cancelation Cleanup Handlers

The cancelation cleanup handlers provide a portable mechanism, easy to implement, for releasing resources and restoring invariants. They are easier to use than signal handlers because they provide a stack of cancelation cleanup handlers rather than a single handler, and because they have an argument that can be used to pass context information to the handler.

The alternative to providing these simple cancelation cleanup handlers (whose only use is for cleaning up when a thread is canceled) is to define a general exception package that could be used for handling and cleaning up after hardware traps and software detected errors. This was too far removed from the charter of providing threads to handle asynchrony. However, it is an explicit goal of IEEE Std. 1003.1-200x to be compatible with existing exception facilities and languages having exceptions.

The interaction of this facility and other procedure-based or language-level exception facilities is unspecified in this version of IEEE Std. 1003.1-200x. However, it is intended that it be possible for an implementation to define the relationship between these cancelation cleanup handlers and Ada, C++, or other language-level exception handling facilities.

It was suggested that the cancelation cleanup handlers should also be called when the process exits or calls the *exec* function. This was rejected partly due to the performance problem caused by having to call the cancelation cleanup handlers of every thread before the operation could continue. The other reason was that the only state expected to be cleaned up by the cancelation cleanup handlers would be the intraprocess state. Any handlers that are to clean up the interprocess state would be registered with *atexit*( ). There is the orthogonal problem that the *exec* functions do not honor the *atexit*( ) handlers, but resolving this is beyond the scope of IEEE Std. 1003.1-200x.

• Async-Cancel Safety

A function is said to be *async-cancel safe* if it is written in such a way that entering the function with asynchronous cancelability enabled will not cause any invariants to be violated, even if a cancelation request is delivered at any arbitrary instruction. Functions that are async-cancel-safe are often written in such a way that they need to acquire no resources for their operation and the visible variables that they may write are strictly limited.

Any routine that gets a resource as a side-effect cannot be made async-cancel-safe (for example, *malloc*( )). If such a routine were called with asynchronous cancelability enabled, it might acquire the resource successfully, but as it was returning to the client, it could act on a cancelation request. In such a case, the application would have no way of knowing whether the resource was acquired or not.

Indeed, because many interesting routines cannot be made async-cancel-safe, most library routines in general are not async-cancel-safe. Every library routine should specify whether or not it is async-cancel safe so that programmers know which routines can be called from code that is asynchronously cancelable.

B.2.9.6 Thread Read-Write Locks

Background

Read-write locks are often used to allow parallel access to data on multi-processors, to avoid context switches on uni-processors when multiple threads access the same data, and to protect data structures that are frequently accessed (that is, read) but rarely updated (that is, written). The in-core representation of a file system directory is a good example of such a data structure. One would like to achieve as much concurrency as possible when searching directories, but limit concurrent access when adding or deleting files.

Although read-write locks can be implemented with mutexes and condition variables, such implementations are significantly less efficient than is possible. Therefore, this synchronization primitive is included in IEEE Std. 1003.1-200x for the purpose of allowing more efficient implementations in multi-processor systems.

Queuing of Waiting Threads

The *pthread_rwlock_unlock*( ) function description states that one writer or one or more readers shall acquire the lock if it is no longer held by any thread as a result of the call. However, the function does not specify which thread(s) acquire the lock, unless the Thread Execution Scheduling option is supported.

The standard developers considered the issue of scheduling with respect to the queuing of threads blocked on a read-write lock. The question turned out to be whether IEEE Std. 1003.1-200x should require priority scheduling of read-write locks for threads whose execution scheduling policy is priority-based (for example, SCHED_FIFO or SCHED_RR). There are tradeoffs between priority scheduling, the amount of concurrency achievable among readers, and the prevention of writer and/or reader starvation.

For example, suppose one or more readers hold a read-write lock and the following threads request the lock in the listed order:

```
pthread_rwlock_wrlock() - Low priority thread writer_a
pthread_rwlock_rdlock() - High priority thread reader_a
pthread_rwlock_rdlock() - High priority thread reader_b
pthread_rwlock_rdlock() - High priority thread reader_c
```

When the lock becomes available, should *writer_a* block the high priority readers? Or, suppose a read-write lock becomes available and the following are queued:

```
pthread_rwlock_rdlock() - Low priority thread reader_a
pthread_rwlock_rdlock() - Low priority thread reader_b
pthread_rwlock_rdlock() - Low priority thread reader_c
pthread_rwlock_wrlock() - Medium priority thread writer_a
pthread_rwlock_rdlock() - High priority thread reader_d
```

If priority scheduling is applied then *reader_d* would acquire the lock and *writer_a* would block the remaining readers. But should the remaining readers also acquire the lock to increase concurrency? The solution adopted takes into account that when the Thread Execution Scheduling option is supported, high priority threads may in fact starve low priority threads (the application developer is responsible in this case to design the system in such a way that this starvation is avoided). Therefore, IEEE Std. 1003.1-200x specifies that high priority readers take precedence over lower priority writers. However, to prevent writer starvation from threads of the same or lower priority, writers take precedence over readers of the same or lower priority.

Priority inheritance mechanisms are non-trivial in the context of read-write locks. When a high priority writer is forced to wait for multiple readers, for example, it is not clear which subset of the readers should inherit the writer's priority. Furthermore, the internal data structures that record the inheritance must be accessible to all readers, and this implies some sort of serialization that could negate any gain in parallelism achieved through the use of multiple readers in the first place. Finally, existing practice does not support the use of priority inheritance for read-write locks. Therefore, no specification of priority inheritance or priority ceiling is attempted. If reliable priority-scheduled synchronization is absolutely required, it can always be obtained through the use of mutexes.

Comparison to fcntl() Locks

The read-write locks and the *fcntl*( ) locks in IEEE Std. 1003.1-200x share a common goal: increasing concurrency among readers, thus increasing throughput and decreasing delay.

However, the read-write locks have two features not present in the *fcntl*( ) locks. First, under priority scheduling, read-write locks are granted in priority order. Second, also under priority scheduling, writer starvation is prevented by giving writers preference over readers of equal or lower priority.

Also, read-write locks can be used in systems lacking a file system, such as those conforming to the minimal realtime system profile of IEEE Std. 1003.13-1998.

History of Resolution Issues

Based upon some balloting objections, the draft specified the behavior of threads waiting on a read-write lock during the execution of a signal handler, as if the thread had not called the lock operation. However, this specified behavior would require implementations to establish internal signal handlers even though this situation would be rare, or never happen for many programs. This would introduce an unacceptable performance hit in comparison to the little additional functionality gained. Therefore, the behavior of read-write locks and signals was reverted back to its previous mutex-like specification.

B.2.9.7 Thread Interactions with Regular File Operations

There is no additional rationale for this section.

B.2.10 Sockets

There is no additional rationale for this section.

B.2.10.1 Protocol Families

There is no additional rationale for this section.

B.2.10.2 Protocols

There is no additional rationale for this section.

B.2.10.3 Addressing

There is no additional rationale for this section.

B.2.10.4 Routing

There is no additional rationale for this section.

B.2.10.5 Interfaces

There is no additional rationale for this section.

B.2.10.6 Socket Types

There is no additional rationale for this section.

B.2.10.7 Socket I/O Mode

There is no additional rationale for this section.

B.2.10.8 Socket Owner

There is no additional rationale for this section.

B.2.10.9 Socket Queue Limits

There is no additional rationale for this section.

B.2.10.10 Pending Error

There is no additional rationale for this section.

B.2.10.11 Socket Receive Queue

There is no additional rationale for this section.

B.2.10.12 Socket Out-of-Band Data State

There is no additional rationale for this section.

B.2.10.13 Connection Indication Queue

There is no additional rationale for this section.

B.2.10.14 Signals

There is no additional rationale for this section.

B.2.10.15 Asynchronous Errors

There is no additional rationale for this section.

B.2.10.16 Use of Options

There is no additional rationale for this section.

B.2.10.17 Use of Sockets for Local UNIX Connections

There is no additional rationale for this section.

B.2.10.18 Use of Sockets over Internet Protocols Based on IPv4

There is no additional rationale for this section.

B.2.10.19 Use of Sockets over Internet Protocols Based on IPv6

There is no additional rationale for this section.

B.2.11 Tracing

The organization of the tracing rationale differs from the traditional rationale in that this tracing rationale text is written against the trace interface as a whole, rather than against the individual components of the trace interface or the normative section in which those components are defined. Therefore the sections below do not parallel the sections of normative text in IEEE Std. 1003.1-200x.

B.2.11.1 Objectives

The intended uses of tracing are application-system debugging during system development, as a ‘‘flight recorder’’ for maintenance of fielded systems, and as a performance measurement tool. In all of these intended uses, the vendor-supplied computer system and its software are, for this discussion, assumed error-free; the intent being to debug the user-written and/or third-party application code, and their interactions. Clearly, problems with the vendor-supplied system and its software will be uncovered from time to time, but this is a byproduct of the primary activity, debugging user code.

Another need for defining a trace interface in POSIX stems from the objective to provide an efficient portable way to perform benchmarks. Existing practice shows that such interfaces are commonly used in a variety of systems but with little commonality. As part of the benchmarking needs, we must consider two aspects within the trace interface.

The first, and perhaps more important one, is the qualitative aspect.

The second is the quantitative aspect.

- Qualitative Aspect

  To better understand this aspect, let us consider an example. Suppose that you want to organize a number of actions to be performed during the day. Some of these actions are known at the beginning of the day. Some others, which may be more or less important, will be triggered by reading your mail. During the day you will make some phone calls and synchronously receive some more information. Finally you will receive asynchronous phone calls that also will trigger actions. If you, or somebody else, examines your day at work, you, or he, can discover that you have not efficiently organized your work. For instance, relative to the phone calls you made, would it be preferable to make some of these early in the morning? Or to delay some others until the end of the day? Relative to the phone calls you have received, you might find that somebody you called in the morning has called you 10 times while you were performing some important work. To examine, afterwards, your day at work, you record in sequence all the trace events relative to your work. This should give you a chance of organizing your next day at work.

This is the qualitative aspect of the trace interface. The user of a system needs to keep a trace of particular points the application passes through, so that he can eventually make some changes in the application and/or system configuration, to give the application a chance of running more efficiently.

- Quantitative Aspect

  This aspect concerns primarily realtime applications, where missed deadlines can be undesirable. Although there are, in POSIX.1b and POSIX.1c/POSIX.1d/POSIX.1j, some interfaces useful for such applications (timeouts, execution time monitoring, and so on), there are no APIs to aid in the tuning of a realtime application's behavior (**timespec** in timeouts, length of message queues, duration of driver interrupt service routine, and so on). The tuning of an application needs a means of recording timestamped important trace events during execution in order to analyze offline, and eventually, to tune some realtime features (redesign the system with less functionalities, readjust timeouts, redesign driver interrupts, and so on).

Detailed Objectives

Objectives were defined to build the trace interface and are kept for historical interest. Although some objectives are not fully respected in this trace interface, the concept of the POSIX trace interface assumes the following points:

1. It shall be possible to trace both system and user trace events concurrently.
2. It must be possible to trace per-process trace events and also to trace system trace events which are unrelated to any particular process. A per-process trace event is either user-initiated or system-initiated.
3. It must be possible to control tracing on a per process basis from either inside or outside the process.
4. It must be possible to control tracing on a per-thread basis from inside the enclosing process.
5. Trace points shall be controllable by trace event type ID from inside and outside of the process. Multiple trace points can have the same trace event type ID, and will be controlled jointly.
6. Recording of trace events is dependent on both trace event type ID and the process/thread. Both must be enabled in order to record trace events. System trace events may or may not be handled differently.
7. The API shall not mandate the ability to control tracing for more than one process at the same time.

8. There is no objective for trace control on anything bigger than a process; for example, group or session.

9. Trace propagation and control:

   a. Trace propagation across fork is optional; the default is to not trace a child process.

   b. Trace control shall span *thread_create* operations; that is, if a process is being traced, any thread will be traced as well if this thread allows tracing. The default is to allow tracing.

10. Trace control shall not span *exec* or *spawn* operations.

11. A triggering API is not required. The triggering API is the ability to command or stop tracing based on the occurrence of specific trace event other than a POSIX_TRACE_START trace event or a POSIX_TRACE_STOP trace event.

12. Trace log entries shall have timestamps of implementation-defined resolution. Implementations are exhorted to support at least microsecond resolution. When a trace log entry is retrieved, it shall have timestamp, PC address, PID, and TID of the entity that generated the trace event.

13. Independently developed code should be able to use trace facilities without coordination and without conflict.

14. Even if the trace points in the trace calls are not unique, the trace log entries (after any processing) shall be uniquely identified as to trace point.

15. There shall be a standard API to read the trace stream.

16. The format of the trace stream and the trace log is opaque and unspecified.

17. It shall be possible to read a completed trace, if recorded on some suitable non-volatile storage, even subsequent to a power cycle or subsequent cold boot of the system.

18. Support of analysis of a trace log while it is being formed is implementation-defined.

19. The API shall allow the application to write trace stream identification information into the trace stream and to be able to retrieve it, without it being overwritten by trace entries, even if the trace stream is full.

20. It must be possible to specify the destination of trace data produced by trace events.

21. It must be possible to have different trace streams, and for the tracing enabled by one trace stream to be completely independent of the tracing of another trace stream.

22. It must be possible to trace events from threads in different CPUs.

23. The API shall support one or more trace streams per-system, and one or more trace streams per-process, up to an implementation-defined set of per-system and per-process maximums.

24. It shall be possible to determine the order in which the trace events happened, without necessarily depending on the clock, up to an implementation-defined time resolution.

25. For performance reasons, the trace event point call(s) shall be implementable as a macro (see the ISO POSIX-1:1996 standard, Subclause 1.3.4, Statement 2).

26. IEEE Std. 1003.1-200x must not define the trace points which a conforming system must implement, except for trace points used in the control of tracing.

27. The APIs shall be thread-safe, and trace points should be lock-free (that is shall not require a lock to gain exclusive access to some resource).

28. The user-provided information associated with a trace event is variable-sized, up to some maximum size.

29. Bounds on record and trace stream sizes:

    a. The API must permit the application to declare the upper bounds on the length of an application data record. The system shall return the limit it used. The limit used may be smaller than requested.

    b. The API must permit the application to declare the upper bounds on the size of trace streams. The system shall return the limit it used. The limit used may be different, either larger or smaller, than requested.

30. The API must be able to pass any fundamental data type, and a structured data type composed only of fundamental types. The API must be able to pass data by reference, given only as an address and a length. Fundamental types are the POSIX.1 types (see the ISO POSIX-1: 1996 standard, Subclause 2.5, Table 2-1) plus those defined in the ISO C standard.

31. The API shall apply the POSIX notions of ownership and permission to recorded trace data, corresponding to the sources of that data.

Comments on Objectives

Note: In the following comments, numbers in square brackets refer to the above objectives.

It is necessary to be able to obtain a trace stream for a complete activity. This means we need to be able to trace both application and system trace events. A per-process trace event is either user-initiated, like the *write*( ) POSIX call, or system-initiated, like a timer expiration. We also need to be able to trace an entire process's activity even when it has threads in multiple CPUs. To avoid excess trace activity, it is necessary to be able to control tracing on a trace event type basis.
[Objectives 1,2,5,22]

We need to be able to control tracing on a per-process basis, both from inside and outside the process; that is, a process can start a trace activity on itself or any other process. We also see the need to allow the definition of a maximum number trace streams per system.
[Objectives 3,23]

From within a process, it is necessary to be able to control tracing on a per-thread basis. This provides an additional filtering capability to keep the amount of traced data to a minimum. It also allows for less ambiguity as to the origin of trace events. It is recognized that thread-level control is only valid from within the process itself. It is also desirable to know the maximum number of trace streams per process that can be started. We do not want the API to require thread synchronization or to mandate priority inversions that would cause the thread to block. However, the API must be thread-safe.
[Objectives 4,23,24,27]

We see no objective to control tracing on anything larger than a process; for example, a group or session. Also, the ability to start or stop a trace activity on multiple processes atomically may be very difficult or cumbersome in some implementations.
[Objectives 6,8]

It is also necessary to be able to control tracing by trace event type identifier, sometimes called a trace hook ID. However, there is no mandated set of system trace events, since such trace points are very system-dependent. The API must not require from the operating system facilities that are not standard (POSIX).
[Objectives 6,26]

Trace control must span *fork*( ) and *pthread_create*( ). If not, there will be no way to ensure that a program's activity is entirely traced. The newly forked child would not be able to turn on its tracing until after it obtained control after the fork, and trace control externally would be even more problematic.
[Objective 9]

Since *exec*( ) and *spawn*( ) represent a complete change in the execution of a task (a new program), trace control need not persist over an *exec*( ) or *spawn*( ).
[Objective 10]

Where trace activities are started on multiple processes, these trace activities should not interfere with each other.
[Objective 21]

There is no need for a triggering objective, primarily for performance reasons; see also Section B.2.11.8 (on page 3498), rationale on triggering.
[Objective 11]

It must be possible to determine the origin of each traced event. We need the process and thread identifiers for each trace event. We also saw the need for a user-specifiable origin, but felt this would create too much overhead.
[Objectives 12,14]

We must allow for trace points to come embedded in software components from several different sources and vendors without requiring coordination.
[Objective 13]

We need to be able to uniquely identify trace points that may have the same trace stream identifier. We only need to be able to do this when a trace report is produced.
[Objectives 12,14]

Tracing is a very performance-sensitive activity, and will therefore likely be implemented at a low level within the system. Hence the interface shall not mandate any particular buffering or storage method. Therefore, we will need a standard API to read a trace stream. Also the interface shall not mandate the format of the trace data, and the interface shall not assume a trace storage method. Due to the possibility of a monolithic kernel and the possible presence of multiple processes capable of running trace activities, the two kinds of trace events may be stored in two separate streams for performance reasons. A mandatory dump mechanism, common in some existing practice, has been avoided to allow the implementation of this set of functions on small realtime profiles for which the concept of a file system is not defined. The trace API calls should be implemented as macros.
[Objectives 15,16,25,30]

Since a trace facility is a valuable service tool, the output (or log) of a completed trace stream that is written to permanent storage must be readable on other systems of the type that produced the trace log. Note that there is no objective to be able to interpret a trace log that was not successfully completed.
[Objectives 17,18,19]

For trace streams written to permanent storage, a way to specify the destination of the trace stream is needed.
[Objective 20]

We need to be able to depend on the ordering of trace events up to some system-defined time interval. For example, we need to know the time period which, if trace events are closer together, their ordering is indeterminate. Events that occur within an interval smaller than this resolution may or may not be read back in the correct order.

[Objective 24]

The application should be able to know how much data can be traced. When trace event types can be filtered, the application should be able to specify the approximate maximum amount of data that will be traced in a trace event so resources can be more efficiently allocated. [Objectives 28,29]

Users should not be able to trace data to which they would not normally have access to. System trace events corresponding to a process / thread should be associated with the ownership of that process / thread. [Objective 31]

B.2.11.2 Trace Model

Introduction

The model is based on two base entities: the ''Trace Stream'' and the ''Trace Log'' , and a recorded unit called the ''Trace Event''. The possibility of using Trace Streams and Trace Logs separately gives us two use dimensions and solves both the performance issue and the full-information system issue. In the case of a trace stream without log, specific information, although reduced in quantity, is required to be registered, in a possibly small realtime system, with as little overhead as possible. The Trace Log option has been added for small realtime systems. In the case of a trace stream with log, considerable complex application-specific information needs to be collected.

Trace Model Description

The trace model can be examined for three different subfunctions: Application Instrumentation, Trace Operation Control, and Trace Analysis.

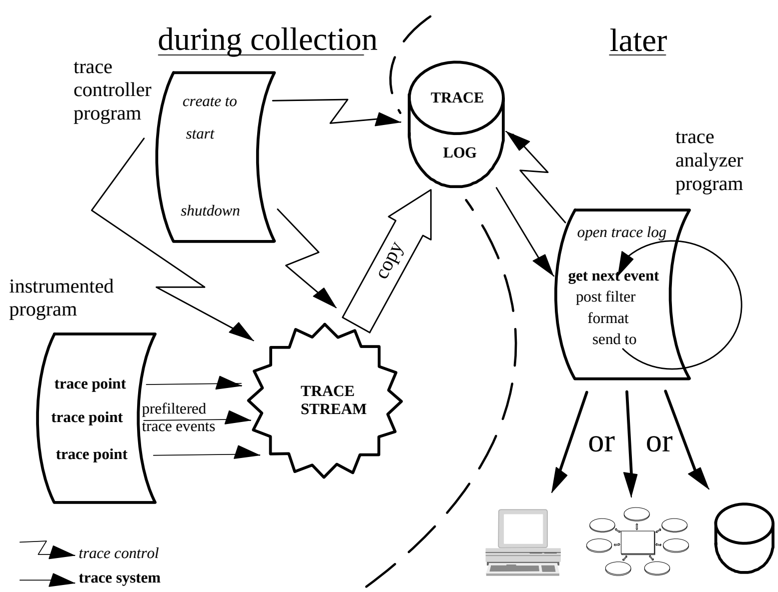


Figure B-2 Trace System Overview: for Offline Analysis

Each of these subfunctions requires specific characteristics of the trace mechanism API.

- Application Instrumentation

  When instrumenting an application, the programmer has no concern about the future utilization of the trace events in trace stream or trace log, the full policy of trace stream, or the eventual pre-filtering of trace events. But he is concerned about the correct determination of specific trace event type identifier, regardless of how many independent libraries are used in the same user application; see Figure B-2 and Figure B-3 (on page 3482).

  This trace API shall provide the necessary operations to accomplish this subfunction. This is done by providing functions to associate a programmer-defined name with an implementation-defined trace event type identifier; see the *posix_trace_eventid_open*( ) function), and to send this trace event into a potential trace stream (see the *posix_trace_event*( ) function).

- Trace Operation Control

  When controlling the recording of trace events in a trace stream, the programmer is concerned with the correct initialization of the trace mechanism (that is, the sizing of the trace stream), the correct retention of trace events in a permanent storage, the correct dynamic recording of trace events, and so on.

  This trace API shall provide the necessary material to permit this efficiently. This is done by providing functions to initialize a new trace stream, and optionally a trace log:

  — Trace Stream Attributes Object Initialization (see *posix_trace_attr_init*( ))

  — Functions to Retrieve or Set Information About a Trace Stream (see *posix_trace_attr_getgenversion*( ))

  — Functions to Retrieve or Set the Behavior of a Trace Stream (see *posix_trace_attr_getinherited*( ))

  — Functions to Retrieve or Set Trace Stream Size Attributes (see *posix_trace_attr_getmaxusereventsize*( ))

  — Trace Stream Initialization, Flush, and Shutdown from a Process (see *posix_trace_create*( ))

  — Clear Trace Stream and Trace Log (see *posix_trace_clear*( ))

  To select the trace event types that are to be traced:

  — Manipulate Trace Event Type Identifier (see *posix_trace_trid_eventid_open*( ))

  — Iterate over a Mapping of Trace Event Type (see *posix_trace_eventtypelist_getnext_id*( ))

  — Manipulate Trace Event Type Sets (see *posix_trace_eventset_empty*( ))

  — Set Filter of an Initialized Trace Stream (see *posix_trace_set_filter*( ))

  To control the execution of an active trace stream:

  — Trace Start and Stop (see *posix_trace_start*( ))

  — Functions to Retrieve the Trace Attributes or Trace Statuses (see *posix_trace_get_attr*( ))

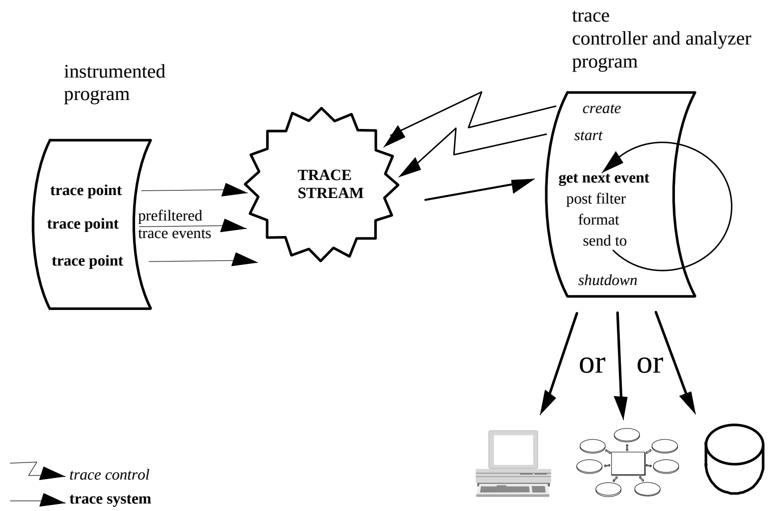


Figure B-3 Trace System Overview: for Online Analysis

- Trace Analysis

  Once correctly recorded, on permanent storage or not, an ultimate activity consists of the analysis of the recorded information. If the recorded data is on permanent storage, a specific open operation is required to associate a trace stream to a trace log.

  The first intent of the group was to request the presence of a system identification structure in the trace stream attribute. This was, for the application, to allow some portable way to process the recorded information. However, there is no requirement that the **utsname** structure, on which this system identification was based, be portable from one machine to another, so the contents of the attribute cannot be interpreted correctly by an application conforming to IEEE Std. 1003.1-200x.

  Draft 6 incorporates this modification and requests that some unspecified information be recorded in the trace log in order to fail opening it if the analysis process and the controller process were running in different types of machine, but does not request that this information be accessible to the application. This modification has implied a modification in the *posix_trace_open*( ) function error code returns.

  This trace API shall provide functions to:

  — Extract trace stream identification attributes (see *posix_trace_attr_getgenversion*( ))

  — Extract trace stream behavior attributes (see *posix_trace_attr_getinherited*( ))

  — Extract trace event, stream, and log size attributes (see *posix_trace_attr_getmaxusereventsize*( ))

  — Look up trace event type names (see *posix_trace_eventid_get_name*( ))

— Iterate over trace event type identifiers (see *posix_trace_eventtypelist_getnext_id*( ))

— Open, rewind, and close a trace log (see *posix_trace_open*( ))

— Read trace stream attributes and status (see *posix_trace_get_attr*( ))

— Read trace events (see *posix_trace_getnext_event*( ))

Due to the following two reasons:

1. The requirement that the trace system must not add unacceptable overhead to the traced process and so that the trace event point execution must be fast

2. The traced application does not care about tracing errors

the trace system cannot return any internal error to the application. Internal error conditions can range from unrecoverable errors that will force the active trace stream to abort, to small errors that can affect the quality of tracing without aborting the trace stream. The group decided to define a system trace event to report to the analysis process such internal errors. It is not the intention of IEEE Std. 1003.1-200x to require an implementation to report an internal error that corrupts or terminates tracing operation. The implementor is free to decide which internal documented errors, if any, the trace system is able to report.

States of a Trace Stream

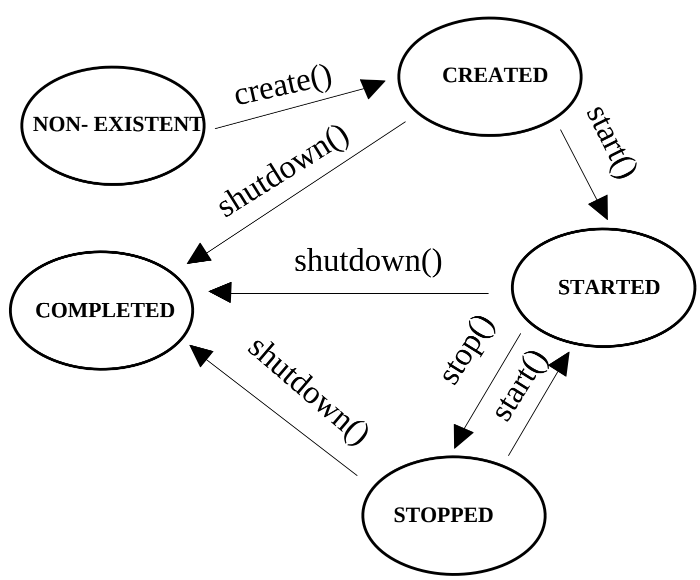


Figure B-4 Trace System Overview: States of a Trace Stream

Figure B-4 shows the different states an active trace stream passes through. After the *posix_trace_create*( ) function call, a trace stream becomes CREATED and a trace stream is associated for the future collection of trace events. The status of the trace stream is POSIX_TRACE_SUSPENDED. The state becomes STARTED after a call to the *posix_trace_start*( ) function, and the status becomes POSIX_TRACE_RUNNING. In this state, all trace events that are not filtered out shall be stored into the trace stream. After a call to *posix_trace_stop*( ), the trace stream becomes STOPPED (and the status POSIX_TRACE_SUSPENDED). In this state, no new trace events will be recorded in the trace stream, but previously recorded trace events may continue to be read.

After a call to *posix_trace_shutdown*( ), the trace stream is in the state COMPLETED. The trace stream no longer exists but, if the Trace Log option is supported, all the information contained in it has been logged. If a log object has not been associated with the trace stream at the creation, it is the responsibility of the trace controller process to not shut the trace stream down while trace events remain to be read in the stream.

Tracing All Processes

Some implementations have a tracing subsystem with the ability to trace all processes. This is useful to debug some types of device drivers such as those for ATM or X25 adapters. These types of adapters are used by several independent processes, that are not issued from the same process.

The POSIX trace interface does not define any constant or option to create a trace stream tracing all processes. But the POSIX trace interface does not prevent this type of implementation and the implementor is free to add this capability. Nevertheless, the POSIX trace interface allows to trace all the system trace events and all the processes issued from the same process.

If such a tracing system capability has to be implemented, when a trace stream is created, it is recommended that a constant named POSIX_TRACE_ALLPROC be used instead of the process identifier in the argument of the function *posix_trace_create*( ) or *posix_trace_create_withlog*( ). A possible value for POSIX_TRACE_ALLPROC may be −1 instead of a real process identifier.

The implementor has to be aware that there is some impact on the tracing behavior as defined in the POSIX trace interface. For example:

- If the default value for the inheritance attribute is to set to POSIX_TRACE_CLOSE_FOR_CHILD, the implementation has to stop tracing for the child process.
- The trace controller which is creating this type of trace stream must have the appropriate privilege to trace all the processes.

Trace Storage

The model is based on two types of trace events: system trace events and user-defined trace events. The internal representation of trace events is implementation-defined, and so the implementor is free to choose the more suitable, practical, and efficient way to design the internal management of trace events. For the timestamping operation, the model does not impose the CLOCK_REALTIME or any other clock. The buffering allocation and operation follow the same principle. The implementor is free to use one or more buffers to record trace events; the interface assumes only a logical trace stream of sequentially recorded trace events. Regarding flushing of trace events, the interface allows the definition of a trace log object which typically can be a file. But the group was also aware of defining functions to permit the use of this interface in small realtime systems, which may not have general file system capabilities. For instance, the three functions *posix_trace_getnext_event*( ) (blocking), *posix_trace_timedgetnext_event*( ) (blocking with timeout), and *posix_trace_trygetnext_event*( ) (non-blocking) are proposed to read the recorded trace events.

The policy to be used when the trace stream becomes full also relies on common practice:

- For an active trace stream, the POSIX_TRACE_LOOP trace stream policy permits automatic overrun (overwrite of oldest trace events) while waiting for some user-defined condition to cause tracing to stop. By contrast, the POSIX_TRACE_UNTIL_FULL trace stream policy requires the system to stop tracing when the trace stream is full. However, if the trace stream that is full is at least partially emptied by a call to the *posix_trace_flush*( ) function or by calls to *posix_trace_getnext_event*( ) function, the trace system will automatically resume tracing.

If the Trace Log option is supported the operation of the POSIX_TRACE_FLUSH policy is an extension of the POSIX_TRACE_UNTIL_FULL policy. The automatic free operation (by flushing to the associated trace log) is added.

- If a log is associated with the trace stream and this log is a regular file, these policies also apply for the log. One more policy, POSIX_TRACE_APPEND, is defined to allow indefinite extension of the log. Since the log destination can be any device or pseudo-device, the implementation may not be able to manipulate the destination as required by IEEE Std. 1003.1-200x. For this reason, the behavior of the log full policy may be unspecified depending of the trace log type.

  The current trace interface does not define a service to preallocate space for a trace log file, because this space can be preallocated by means of a call to the *posix_fallocate*( ) function. This function could be called after the file has been opened, but before the trace stream is created. The *posix_fallocate*( ) function ensures that any required storage for regular file data is allocated on the file system storage media. If *posix_fallocate*( ) returns successfully, subsequent writes to the specified file data shall not fail due to the lack of free space on the file system storage media. Besides trace events, a trace stream also includes trace attributes and the mapping from trace event names to trace event type identifiers. The implementor is free to choose how to store the trace attributes and the trace event type map, but must ensure that this information is not lost when a trace stream overrun occurs.

B.2.11.3 Trace Programming Examples

Several programming examples are presented to show the code of the different possible subfunctions using a trace subsystem. All these programs need to include the **<trace.h>** header. In the examples shown, error checking is omitted for more simplicity.

Trace Operation Control

These examples show the creation of a trace stream for another process; one which is already trace instrumented. All the default trace stream attributes are used to simplify programming in the first example. The second example shows more possibilities.

First Example

```
/* Caution. Error checks omitted */
{
    trace_attr_t attr;
    pid_t pid = traced_process_pid;
    int fd;
    trace_id_t trid;

    - - - - - -
    /* Initialize trace stream attributes */
    posix_trace_attr_init(&attr);
    /* Open a trace log */
    fd=open("/tmp/mytracelog",...);
    /*
     * Create a new trace associated with a log
     * and with default attributes
```

```
         */
        posix_trace_create_withlog(pid, &attr, fd, &trid);
        /* Trace attribute structure can now be destroyed */
        posix_trace_attr_destroy(&attr);
        /* Start of trace event recording */
        posix_trace_start(trid);
        - - - - - -
        - - - - - -
        /* Duration of tracing */
        - - - - - -
        - - - - - -
        /* Stop and shutdown of trace activity */
        posix_trace_shutdown(trid);
        - - - - - -
    }
```

Second Example

Between the initialization of the trace stream attributes and the creation of the trace stream, these trace stream attributes may be modified; see **Trace Stream Attribute Manipulation** (on page 3490) for specific programming example. Between the creation and the start of the trace stream, the event filter may be set; after the trace stream is started, the event filter may be changed. The setting of an event set and the change of a filter is shown in **Create a Trace Event Type Set and Change the Trace Event Type Filter** (on page 3490).

```
/* Caution. Error checks omitted */
{
    trace_attr_t attr;
    pid_t pid = traced_process_pid;
    int fd;
    trace_id_t trid;
    - - - - - -
    /* Initialize trace stream attributes */
    posix_trace_attr_init(&attr);
    /* Attr default may be changed at this place; see example */
    - - - - - -
    /* Create and open a trace log with R/W user access */
    fd=open("/tmp/mytracelog",O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
    /* Create a new trace associated with a log */
    posix_trace_create_withlog(pid, &attr, fd, &trid);
    /*
     * If the Trace Filter option is supported
     * trace event type filter default may be changed at this place;
     * see example about changing the trace event type filter
     */
    posix_trace_start(trid);
    - - - - - -

    /*
     * If you have an uninteresting part of the application
     * you can stop temporarily.
     *
```

```
             * posix_trace_stop(trid);
             * - - - - - -
             * - - - - - -
             * posix_trace_start(trid);
             */
            - - - - - -
            /*
             * If the Trace Filter option is supported
             * the current trace event type filter can be changed
             * at any time (see example about how to set
             * a trace event type filter
             */
            - - - - - -
            /* Stop the recording of trace events */
            posix_trace_stop(trid);
            /* Shutdown the trace stream */
            posix_trace_shutdown(trid);
            /*
             * Destroy trace stream attributes; attr structure may have
             * been used during tracing to fetch the attributes
             */
            posix_trace_attr_destroy(&attr);
            - - - - - -
        }
```

Application Instrumentation

This example shows an instrumented application. The code is included in a block of instructions, perhaps a function from a library. Possibly in an initialization part of the instrumented application, two user trace events names are mapped to two trace event type identifiers (function *posix_trace_eventid_open*( )). Then two trace points are programmed.

```
/* Caution. Error checks omitted */
{
    trace_eventid_t eventid1, eventid2;
    - - - - - -
    /* Initialization of two trace event type ids */
    posix_trace_eventid_open("my_first_event",&eventid1);
    posix_trace_eventid_open("my_second_event",&eventid2);
    - - - - - -
    - - - - - -
    - - - - - -
    /* Trace point */
    posix_trace_event(eventid1,NULL,0);
    - - - - - -
    /* Trace point */
    posix_trace_event(eventid2,NULL,0);
    - - - - - -
}
```

Trace Analyzer

This example shows the manipulation of a trace log resulting from the dumping of a completed trace stream. All the default attributes are used to simplify programming, and data associated with a trace event are not shown in the first example. The second example shows more possibilities.

First Example

```
/* Caution. Error checks omitted */
{
    int fd;
    trace_id_t trid;
    posix_trace_event_info trace_event;
    char trace_event_name[TRACE_EVENT_NAME_MAX];
    int return_value;
    size_t returndatasize;
    int lost_event_number;

    - - - - - -

    /* Open an existing trace log */
    fd=open("/tmp/tracelog", O_RDONLY);
    /* Open a trace stream on the open log */
    posix_trace_open(fd, &trid);
    /* Read a trace event */
    posix_trace_getnext_event(trid, &trace_event,
        NULL, 0, &returndatasize,&return_value);

    /* Read and print all trace event names out in a loop */
    while (return_value == NULL)
    {
        /*
         * Get the name of the trace event associated
         * with trid trace ID
         */
        posix_trace_eventid_get_name(trid, trace_event.event_id,
            trace_event_name);
        /* Print the trace event name out */
        printf("%s\n",trace_event_name);
        /* Read a trace event */
        posix_trace_getnext_event(trid, &trace_event,
            NULL, 0, &returndatasize,&return_value);
    }

    /* Close the trace stream */
    posix_trace_close(trid);
    /* Close the trace log */
    close(fd);
}
```

Second Example

The complete example includes the two other examples in **Retrieve Information from a Trace Log** (on page 3491) and in **Retrieve the List of Trace Event Types Used in a Trace Log** (on page 3492). For example, the *maxdatasize* variable is set in **Retrieve the List of Trace Event Types Used in a Trace Log** (on page 3492).

```
/* Caution. Error checks omitted */
{
    int fd;
    trace_id_t trid;
    posix_trace_event_info trace_event;
    char trace_event_name[TRACE_EVENT_NAME_MAX];
    char * data;
    size_t maxdatasize=1024, returndatasize;
    int return_value;
    - - - - - -

    /* Open an existing trace log */
    fd=open("/tmp/tracelog", O_RDONLY);
    /* Open a trace stream on the open log */
    posix_trace_open( fd, &trid);
    /*
     * Retrieve information about the trace stream which
     * was dumped in this trace log (see example)
     */
    - - - - - -

    /* Allocate a buffer for trace event data */
    data=(char *)malloc(maxdatasize);
    /*
     * Retrieve the list of trace event used in this
     * trace log (see example)
     */
    - - - - - -

    /* Read and print all trace event names and data out in a loop */
    while (1)
    {
    posix_trace_getnext_event(trid, &trace_event,
        data, maxdatasize, &returndatasize,&return_value);
        if (return_value != NULL) break;
        /*
         * Get the name of the trace event type associated
         * with trid trace ID
         */
        posix_trace_eventid_get_name(trid, trace_event.event_id,
            trace_event_name);
        {
        int i;

        /* Print the trace event name out */
        printf("%s: ", trace_event_name);
        /* Print the trace event data out */
        for (i=0; i<returndatasize, i++) printf("%02.2X",
```

```
            (unsigned char)data[i]);
        printf("\n");
        }
    }
    /* Close the trace stream */
    posix_trace_close(trid);
    /* The buffer data is deallocated */
    free(data);
    /* Now the file can be closed */
    close(fd);
}
```

Several Programming Manipulations

The following examples show some typical sets of operations needed in some contexts.

Trace Stream Attribute Manipulation

This example shows the manipulation of a trace stream attribute object in order to change the default value provided by a previous *posix_trace_attr_init*( ) call.

```
/* Caution. Error checks omitted */
{
    trace_attr_t attr;
    size_t logsize=100000;
    - - - - - -
    /* Initialize trace stream attributes */
    posix_trace_attr_init(&attr);
    /* Set the trace name in the attributes structure */
    posix_trace_attr_setname(&attr, "my_trace");
    /* Set the trace full policy */
    posix_trace_attr_setstreamfullpolicy(&attr, POSIX_TRACE_LOOP);
    /* Set the trace log size */
    posix_trace_attr_setlogsize(&attr, logsize);
    - - - - - -
}
```

Create a Trace Event Type Set and Change the Trace Event Type Filter

This example is valid only if the Trace Event Filter option is supported. This example shows the manipulation of a trace event type set in order to change the trace event type filter for an existing active trace stream, which may be just-created, running, or suspended. Some sets of trace event types are well-known, such as the set of trace event types not associated with a process, some trace event types are just-built trace event types for this trace stream; one trace event type is the predefined trace event error type which is deleted from the trace event type set.

```
/* Caution. Error checks omitted */
{
    trace_id_t trid = existing_trace;
    trace_event_set_t set;
    trace_event_id_t trace_event1, trace_event2;
    - - - - - -
    /* Initialize to an empty set of trace event types */
```

```
        posix_trace_eventset_emptyset(&set);
        /*
         * Fill the set with all system trace events
         * not associated with a process
         */
        posix_trace_eventset_fill(&set, POSIX_TRACE_WOPID_EVENTS);

        /*
         * Get the trace event type identifier of the known trace event name
         * my_first_event for the trid trace stream
         */
        posix_trace_trid_eventid_open(trid, "my_first_event", &trace_event1);
        /* Add the set with this trace event type identifier */
        posix_trace_eventset_add_event(trace_event1, &set);
        /*
         * Get the trace event type identifier of the known trace event name
         * my_second_event for the trid trace stream
         */

        posix_trace_trid_eventid_open(trid, "my_second_event", &trace_event2);
        /* Add the set with this trace event type identifier */
        posix_trace_eventset_add_event(trace_event2, &set);
        - - - - - -
        /* Delete the system trace event POSIX_TRACE_ERROR from the set */
        posix_trace_eventset_del_event(POSIX_TRACE_ERROR, &set);
        - - - - - -

        /* Modify the trace stream filter making it equal to the new set */
        posix_trace_set_filter(trid, &set, POSIX_TRACE_SET_EVENTSET);
        - - - - - -
        /*
         * Now trace_event1, trace_event2, and all system trace event types
         * not associated with a process, except for the POSIX_TRACE_ERROR
         * system trace event type, are filtered out of (not recorded in) the
         * existing trace stream.
         */
    }
```

Retrieve Information from a Trace Log

This example shows how to extract information from a trace log, the dump of a trace stream. This code:

- Asks if the trace stream has lost trace events
- Extracts the information about the version of the trace subsystem which generated this trace log
- Retrieves the maximum size of trace event data; this may be used to dynamically allocate an array for extracting trace event data from the trace log without overflow

```
    /* Caution. Error checks omitted */
    {
        struct posix_trace_status_info statusinfo;
        trace_attr_t attr;
        trace_id_t trid = existing_trace;
```

```
    size_t maxdatasize;
    char genversion[TRACE_NAME_MAX];
    - - - - - -
    /* Get the trace stream status */
    posix_trace_get_status(trid, &statusinfo);
    /* Detect an overrun condition */
    if (statusinfo.posix_stream_overrun_status == POSIX_TRACE_OVERRUN)
        printf("trace events have been lost\n");

    /* Get attributes from the trid trace stream */
    posix_trace_get_attr(trid, &attr);
    /* Get the trace generation version from the attributes */
    posix_trace_attr_getgenversion(&attr, genversion);
    /* Print the trace generation version out */
    printf("Information about Trace Generator:%s\n",genversion);

    /* Get the trace event max data size from the attributes */
    posix_trace_attr_getmaxdatasize(&attr, &maxdatasize);
    /* Print the trace event max data size out */
    printf("Maximum size of associated data:%d\n",maxdatasize);
    /* Destroy the trace stream attributes */
    posix_trace_attr_destroy(&attr);
}
```

Retrieve the List of Trace Event Types Used in a Trace Log

This example shows the retrieval of a trace stream's trace event type list. This operation may be very useful if you are interested only in tracking the type of trace events in a trace log.

```
/* Caution. Error checks omitted */
{
    trace_id_t trid = existing_trace;
    trace_event_id_t event_id;
    char event_name[TRACE_EVENT_NAME_MAX];
    int return_value;
    - - - - - -

    /*
     * In a loop print all existing trace event names out
     * for the trid trace stream
     */
    while (1)
    {
        posix_trace_eventtypelist_getnext_id(trid, &event_id
            &return_value);
        if (return_value != NULL) break;
        /*
         * Get the name of the trace event associated
         * with trid trace ID
         */
        posix_trace_eventid_get_name(trid, event_id, event_name);
        /* Print the name out */
        printf("%s\n", event_name);
    }
```

}

B.2.11.4 Rationale on Trace for Debugging

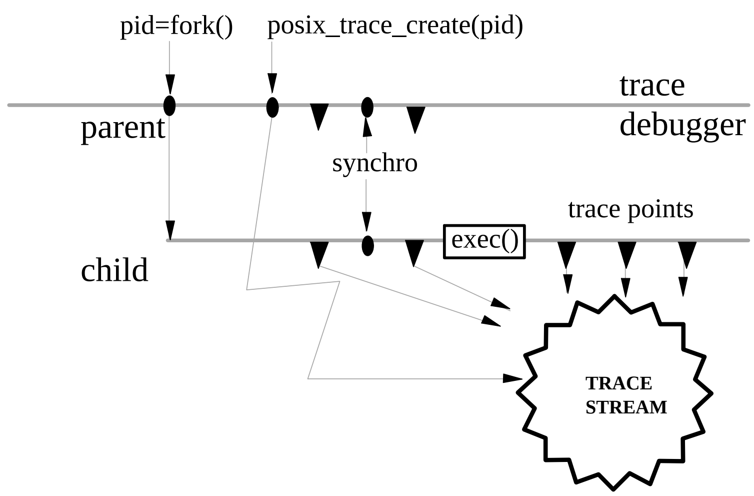


Figure B-5 Trace Another Process

Among the different possibilities offered by the trace interface defined in IEEE Std. 1003.1-200x, the debugging of an application is the most interesting one. Typical operations in the controlling debugger process are to filter trace event types, to get trace events from the trace stream, to stop the trace stream when the debugged process is executing uninteresting code, to start the trace stream when some interesting point is reached, and so on. The interface defined in IEEE Std. 1003.1-200x should define all the necessary base functions to allow this dynamic debug handling.

Figure B-5 shows an example in which the trace stream is created after the call to the *fork*( ) function. If the user does not want to lose trace events some synchronization mechanism (represented in the figure) may be needed before calling the *exec*( ) function, to give the parent a chance to create the trace stream before the child begins the execution of its trace points.

B.2.11.5 Rationale on Trace Event Type Name Space

At first, the working group was in favor of the representation of a trace event type by an integer (*event_name*). It seems that existing practice shows the weakness of such a representation. The collision of trace event types is the main problem that cannot be simply resolved using this sort of representation. Suppose, for example, that a third party designs an instrumented library. The user does not have the source of this library and wants to trace his application which uses in some part the third-party library. There is no means for him to know what are the trace event types used in the instrumented library so he has some chance of duplicating some of them and thus to obtain a contaminated tracing of his application.

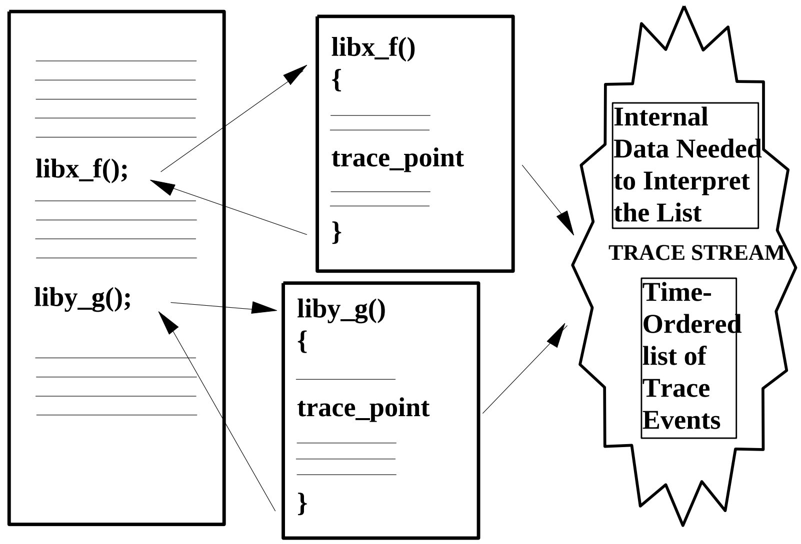


Figure B-6 Trace Name Space Overview: With Third-Party Library

We have requirements to allow program images containing pieces from various vendors to be traced without also requiring those or any other vendors to coordinate their uses of the trace facility, and especially the naming of their various trace event types and trace point IDs. The chosen solution is to provide a very large name space, large enough so that the individual vendors can give their trace types and tracepoint IDs sufficiently long and descriptive names making the occurrence of collisions quite unlikely. The probability of collision is thus made sufficiently low so that the problem may, as a practical matter, be ignored. By requirement, the consequence of collisions will be a slight ambiguity in the trace streams; tracing will continue in spite of collisions and ambiguities. ‘‘The show must go on’’. The *posix_prog_address* member of the **posix_trace_event_info** structure is used to allow trace streams to be unambiguously interpreted, despite the fact that trace event types and trace event names need not be unique.

The *posix_trace_eventid_open*( ) function is required to allow the instrumented third-party library to get a valid trace event type identifier for its trace event names. This operation is, somehow, an allocation, and the group was aware of proposing some deallocation mechanism which the instrumented application could use to recover the resources used by a trace event type identifier. This would have given the instrumented application the benefit of being capable of reusing a possible minimum set of trace event type identifiers, but also the inconvenience to have, possibly in the same trace stream, one trace event type identifier identifying two different trace event types. After some discussions the group decided to not define such a function which would make this API thicker for little benefit, the user having always the possibility of adding identification information in the data member of the trace event structure.

The set of the trace event type identifiers the controlling process wants to filter out is initialized in the trace mechanism using the function *posix_trace_set_filter*( ), setting the arguments according to the definitions explained in *posix_trace_set_filter*( ). This operation can be done statically (when the trace is in the STOPPED state) or dynamically (when the trace is in the STARTED state). The preparation of the filter is normally done using the function defined in *posix_trace_eventtypelist_getnext_id*( ) and eventually the function *posix_trace_eventtypelist_rewind*( ) in order to know (before the recording) the list of the potential set of trace event types that can be recorded. In the case of an active trace stream, this list may not be exhaustive. Actually, the target process may not have yet called the function *posix_trace_eventid_open*( ). But it is a common practice, for a controlling process, to prepare the filtering of a future trace stream before its start. Therefore the user must have a way to get the trace event type identifier corresponding to a well-known trace event name before its future association by the pre-cited function. This is done by calling the *posix_trace_trid_eventid_open*( ) function, given the trace stream identifier and the trace name, and described hereafter. Because this trace event type identifier is associated with a trace stream identifier, where a unique process has initialized two or more traces, the implementation is expected to return the same trace event type identifier for successive calls to *posix_trace_trid_eventid_open*( ) with different trace stream identifiers. The *posix_trace_eventid_get_name*( ) function is used by the controller process to identify, by the name, the trace event type returned by a call to the *posix_trace_eventtypelist_getnext_id*( ) function.

Afterwards, the set of trace event types is constructed using the functions defined in *posix_trace_eventset_empty*( ), *posix_trace_eventset_fill*( ), *posix_trace_eventset_add*( ), and *posix_trace_eventset_del*( ).

A set of functions is provided devoted to the manipulation of the trace event type identifier and names for an active trace stream. All these functions require the trace stream identifier argument as the first parameter. The opacity of the trace event type identifier implies that the user cannot associate directly its well-known trace event name with the system associated trace event type identifier.

The *posix_trace_trid_eventid_open*( ) function allows the application to get the system trace event type identifier back from the system, given its well-known trace event name. One possible use of this function is to qualify a filter.

The *posix_trace_eventid_get_name*( ) function allows the application to obtain a trace event name given its trace event type identifier. One possible use of this function is to identify the type of a trace event retrieved from the trace stream, and print it. The easiest way to implement this requirement, is to use a single trace event type map for all the processes whose maps are required to be identical. A more difficult way is to attempt to keep multiple maps identical at every call to *posix_trace_eventid_open*( ) and *posix_trace_trid_eventid_open*( ).

B.2.11.6 Rationale on Trace Events Type Filtering

The most basic rationale for runtime and pre-registration filtering (selection/rejection) of trace event types is to prevent choking of the trace collection facility, and/or overloading of the computer system. Any worthwhile trace facility can bring even the largest computer to its knees. Otherwise, we would record everything, and filter after the fact; it would be much simpler, but impractical.

To achieve debugging, measurement, or whatever the purpose of tracing, the filtering of trace event types is an important part of trace analysis. Due to the fact that the trace events are put into a trace stream and probably logged afterwards into a file, different levels of filtering—that is, rejection of trace event types—are possible.

Filtering of Trace Event Types Before Tracing

This function, represented by the *posix_trace_set_filter*( ) function in IEEE Std. 1003.1-200x (see *posix_trace_set_filter*( )), selects, before or during tracing, the set of trace event types to be filtered out. It should be possible also (as OSF suggested in their ETAP trace specifications) to select the kernel trace event types to be traced in a system-wide fashion. These two functionalities are called the pre-filtering of trace event types.

The restriction on the actual type used for the **trace_event_set_t** type is intended to guarantee that these objects can always be assigned, have their address taken, and be passed by value as parameters. It is not intended that this type be a structure including pointers to other data structures, as that could impact the portability of applications performing such operations. A reasonable implementation could be a structure containing an array of integer types.

Filtering of Trace Event Types at Runtime

Using this API, this functionality may be built, a privileged process or a privileged thread can get trace events from the trace stream of another process or thread, and thus specify the type of trace events to record into a file, using methods and interfaces out of the scope of IEEE Std. 1003.1-200x. This functionality, called inline filtering of trace event types, is used for runtime analysis of trace streams.

Post-Mortem Filtering of Trace Event Types

The word *post-mortem* is used here to indicate that some unanticipated situation occurs during execution that does not permit a pre or inline filtering of trace events and that it is necessary to record all trace event types, to have a chance to discover the problem afterwards. When the program stops, all the trace events recorded previously can be analyzed in order to find the solution. This functionality could be named the post-filtering of trace event types.

Discussions about Trace Event Type-Filtering

After long discussions with the parties involved in the process of defining the trace interface, it seems that the sensitivity to the filtering problem is different, but everybody agrees that the level of the overhead introduced during the tracing operation depends on the filtering method elected. If the time that it takes the trace event to be recorded can be neglected, the overhead introduced by the filtering process can be classified as follows:

Pre-filtering	System and process / thread-level overhead
Inline-filtering	Process / thread-level overhead
Post-filtering	No overhead; done offline

The pre-filtering could be named *critical realtime* filtering in the sense that the filtering of trace event type is manageable at the user level so the user can lower to a minimum the filtering overhead at some user selected level of priority for the inline filtering, or delay the filtering to after execution for the post-filtering. The counterpart of this solution is that the size of the trace stream must be sufficient to record all the trace events. The advantage of the pre-filtering is that the utilization of the trace stream is optimized.

Only pre-filtering is defined by IEEE Std. 1003.1-200x. However, great care must be taken in specifying pre-filtering, so that it does not impose unacceptable overhead. Moreover, it is necessary to isolate all the functionality relative to the pre-filtering.

The result of this rationale is to define a new option, the Trace Event Filter option, not necessarily implemented in small realtime systems, where system overhead is minimized to the extent possible.

B.2.11.7 Tracing, pthread API

The objective to be able to control tracing for individual threads may be in conflict with the efficiency expected in threads with a contentionscope attribute of PTHREAD_SCOPE_PROCESS. For these threads, context switches from one thread that has tracing enabled to another thread that has tracing disabled may require a kernel call to inform the kernel whether it has to trace system events executed by that thread or not. For this reason, it was proposed that the ability to enable or disable tracing for PTHREAD_SCOPE_PROCESS threads be made optional, through the introduction of a Trace Scope Process option. A trace implementation which did not implement the Trace Scope Process option would not honor the tracing-state attribute of a thread with PTHREAD_SCOPE_PROCESS; it would, however, honor the tracing-state attribute of a thread with PTHREAD_SCOPE_SYSTEM. This proposal was rejected as:

1. Removing desired functionality (per-thread trace control)
2. Introducing counter-intuitive behavior for the tracing-state attribute
3. Mixing logically orthogonal ideas (thread scheduling and thread tracing) [Objective 4]

Finally, to solve this complex issue, this API does not provide *pthread_gettracingstate*( ), *pthread_settracingstate*( ), *pthread_attr_gettracingstate*( ), and *pthread_attr_settracingstate*( ) interfaces. These interfaces force the thread implementation to add to the weight of the thread and cause a revision of the threads libraries, just to support tracing. Worse yet, *posix_trace_userevent*( ) must always test this per-thread variable even in the common case where it is not used at all. Per-thread tracing is easy to implement using existing interfaces where necessary; see the following example.

Example

```
/* Caution. Error checks omitted */
static pthread_key_t my_key;
static trace_event_id_t my_event_id;
static pthread_once_t my_once = PTHREAD_ONCE_INIT;

void my_init(void)
{
    (void) pthread_key_create(&my_key, NULL);
    (void) posix_trace_eventid_open("my", &my_event_id);
}

int get_trace_flag(void)
{
    pthread_once(&my_once, my_init);
    return (pthread_getspecific(my_key) != NULL);
}

void set_trace_flag(int f)
{
    pthread_once(&my_once, my_init);
    pthread_setspecific(my_key, f? &my_event_id: NULL);
}

fn()
{
    if (get_trace_flag())
```

```
        posix_trace_event(my_event_id, ...)
}
```

The above example does not implement third-party state setting, but it is also implementable with some more work, yet the extra functionality is rarely needed.

Lastly, per-thread tracing works poorly for threads with PTHREAD_SCOPE_PROCESS contention scope. These ‘‘library’’ threads have minimal interaction with the kernel and would have to explicitly set the attributes whenever they are context switched to a new kernel thread in order to trace system events. Such state was explicitly avoided in POSIX threads to keep PTHREAD_SCOPE_PROCESS threads lightweight.

The reason that keeping PTHREAD_SCOPE_PROCESS threads lightweight is important is that such threads can be used not just for simple multi-processors but also for coroutine style programming (such as discrete event simulation) without inventing a new threads paradigm. Adding extra runtime cost to thread context switches will make using POSIX threads less attractive in these situations.

B.2.11.8 Rationale on Triggering

The ability to start or stop tracing based on the occurrence of specific trace event types has been proposed as a parallel to similar functionality appearing in logic analyzers. Such triggering, in order to be very useful, should be based not only on the trace event type, but on trace event-specific data, including tests of user-specified fields for matching or threshold values.

Such a facility is unnecessary where the buffering of the stream is not a constraint, since such checks can be performed offline during post-mortem analysis.

For example, a large system could incorporate a daemon utility to collect the trace records from memory buffers and spool them to secondary storage for later analysis. In the instances where resources are truly limited, such as embedded applications, the application incorporation of application code to test the circumstances of a trace event and call the trace point only if needed is usually straightforward.

For performance reasons, the *posix_trace_event*( ) function should be implemented using a macro, so if the trace is inactive, the trace event point calls are latent code and must cost no more than a scalar test.

The API proposed in IEEE Std. 1003.1-200x does not include any triggering functionality.

B.2.11.9 Rationale on Timestamp Clock

It has been suggested that the tracing mechanism should include the possibility of specifying the clock to be used in timestamping the trace events. When application trace events must be correlated to remote trace events, such a facility could provide a global time reference not available from a local clock. Further, the application may be driven by timers based on a clock different from that used for the timestamp, and the correlation of the trace to those untraced timer activities could be an important part of the analysis of the application.

However, the tracing mechanism needs to be fast and just the provision of such an option can materially affect its performance. Leaving aside the performance costs of reading some clocks, this notion is also ill-defined when kernel trace events are to be traced by two applications making use of different tracing clocks. This can even happen within a single application where different parts of the application are served by different clocks. Another complication can occur when a clock is maintained strictly at the user level and is unavailable at the kernel level.

It is felt that the benefits of a selectable trace clock do not match its costs. Applications that wish to correlate clocks other than the default tracing clock can include trace events with sample values of those other clocks, allowing correlation of timestamps from the various independent clocks. In any case, such a technique would be required when applications are sensitive to multiple clocks.

B.2.11.10 Rationale on Different Overrun Conditions

The analysis of the dynamic behavior of the trace mechanism shows that different overrun conditions may occur. The API must provide a means to manage such conditions in a portable way.

Overrun in Trace Streams Initialized with POSIX_TRACE_LOOP Policy

In this case, the user of the trace mechanism is interested in using the trace stream with POSIX_TRACE_LOOP policy to record trace events continuously, but ideally without losing any trace events. The online analyzer process must get the trace events at a mean speed equivalent to the recording speed. Should the trace stream become full, a trace stream overrun occurs. This condition is detected by getting the status of the active trace stream (function *posix_trace_get_status*( )) and looking at the member *posix_stream_overrun_status* of the read **posix_stream_status** structure. In addition, two predefined trace event types are defined:

1. The beginning of a trace overflow, to locate the beginning of an overflow when reading a trace stream
2. The end of a trace overflow, to locate the end of an overflow, when reading a trace stream

As a timestamp is associated with these predefined trace events, it is possible to know the duration of the overflow.

Overrun in Dumping Trace Streams into Trace Logs

The user lets the trace mechanism dump the trace stream initialized with POSIX_TRACE_FLUSH policy automatically into a trace log. If the dump operation is slower than the recording of trace events, the trace stream can overrun. This condition is detected by getting the status of the active trace stream (function *posix_trace_get_status*( )) and looking at the member *posix_log_overrun_status* of the read **posix_stream_status** structure. This overrun indicates that the trace mechanism is not able to operate in this mode at this speed. It is the responsibility of the user to modify one of the trace parameters (the stream size or the trace event type filter, for instance) to avoid such overrun conditions, if overruns are to be prevented. The same already predefined trace event types (see **Overrun in Trace Streams Initialized with POSIX_TRACE_LOOP Policy**) are used to detect and to know the duration of an overflow.

Reading an Active Trace Stream

Although this trace API allows one to read an active trace stream with log while it is tracing, this feature can lead to false overflow origin interpretation: the trace log or the reader of the trace stream. Reading from an active trace stream with log is thus non-portable, and has been left unspecified.

B.2.12 Data Types

The requirement that additional types defined in this section end in ''_t'' was prompted by the problem of name space pollution. It is difficult to define a type (where that type is not one defined by IEEE Std. 1003.1-200x) in one header file and use it in another without adding symbols to the name space of the program. To allow implementors to provide their own types, all conforming applications are required to avoid symbols ending in ''_t'', which permits the implementor to provide additional types. Because a major use of types is in the definition of structure members, which can (and in many cases must) be added to the structures defined in IEEE Std. 1003.1-200x, the need for additional types is compelling.

The types, such as **ushort** and **ulong**, which are in common usage, are not defined in IEEE Std. 1003.1-200x (although **ushort_t** would be permitted as an extension). They can be added to **<sys/types.h>** using a feature test macro (see Section B.2.2.1 (on page 3384)). A suggested symbol for these is _SYSIII. Similarly, the types like **u_short** would probably be best controlled by _BSD.

Some of these symbols may appear in other headers; see Section B.2.2.2 (on page 3384).

dev_t This type may be made large enough to accommodate host-locality considerations of networked systems.

This type must be arithmetic. Earlier proposals allowed this to be non-arithmetic (such as a structure) and provided a *samefile*( ) function for comparison.

gid_t Some implementations had separated **gid_t** from **uid_t** before POSIX.1 was completed. It would be difficult for them to coalesce them when it was unnecessary. Additionally, it is quite possible that user IDs might be different than group IDs because the user ID might wish to span a heterogeneous network, where the group ID might not.

For current implementations, the cost of having a separate **gid_t** will be only lexical.

mode_t This type was chosen so that implementations could choose the appropriate integral type, and for compatibility with the ISO C standard. 4.3 BSD uses **unsigned short** and the SVID uses **ushort**, which is the same. Historically, only the low-order sixteen bits are significant.

nlink_t This type was introduced in place of **short** for *st_nlink* (see the **<sys/stat.h>** header) in response to an objection that **short** was too small.

off_t This type is used only in *lseek*( ), *fcntl*( ), and **<sys/stat.h>**. Many implementations would have difficulties if it were defined as anything other than **long**. Requiring an integral type limits the capabilities of *lseek*( ) to four gigabytes. The ISO C standard supplies routines that use larger types; see *fgetpos*( ) and *fsetpos*( ). XSI-conformant systems provide the *fseeko*( ) and *lseeko*( ) functions that use larger types.

pid_t The inclusion of this symbol was controversial because it is tied to the issue of the representation of a process ID as a number. From the point of view of a portable application, process IDs should be ''magic cookies''[1] that are produced by calls

---

1. An historical term meaning: ''An opaque object, or token, of determinate size, whose significance is known only to the entity which created it. An entity receiving such a token from the generating entity may only make such use of the 'cookie' as is defined and permitted by the supplying entity.''

such as *fork*( ), used by calls such as *waitpid*( ) or *kill*( ), and not otherwise analyzed (except that the sign is used as a flag for certain operations).

The concept of a {PID_MAX} value interacted with this in early proposals. Treating process IDs as an opaque type both removes the requirement for {PID_MAX} and allows systems to be more flexible in providing process IDs that span a large range of values, or a small one.

Since the values in **uid_t**, **gid_t**, and **pid_t** will be numbers generally, and potentially both large in magnitude and sparse, applications that are based on arrays of objects of this type are unlikely to be fully portable in any case. Solutions that treat them as magic cookies will be portable.

{CHILD_MAX} precludes the possibility of a ''toy implementation'', where there would only be one process.

ssize_t This is intended to be a signed analog of **size_t**. The wording is such that an implementation may either choose to use a longer type or simply to use the signed version of the type that underlies **size_t**. All functions that return **ssize_t** (*read*( ) and *write*( )) describe as ''implementation-defined'' the result of an input exceeding {SSIZE_MAX}. It is recognized that some implementations might have **int**s that are smaller than **size_t**. A portable application would be constrained not to perform I/O in pieces larger than {SSIZE_MAX}, but a portable application using extensions would be able to use the full range if the implementation provided an extended range, while still having a single type-compatible interface.

The symbols **size_t** and **ssize_t** are also required in **<unistd.h>** to minimize the changes needed for calls to *read*( ) and *write*( ). Implementors are reminded that it must be possible to include both **<sys/types.h>** and **<unistd.h>** in the same program (in either order) without error.

uid_t Before the addition of this type, the data types used to represent these values varied throughout early proposals. The **<sys/stat.h>** header defined these values as type **short**, the **<passwd.h>** file (now **<pwd.h>** and **<grp.h>**) used an **int**, and *getuid*( ) returned an **int**. In response to a strong objection to the inconsistent definitions, all the types to were switched to **uid_t**.

In practice, those historical implementations that use varying types of this sort can typedef **uid_t** to **short** with no serious consequences.

The problem associated with this change concerns object compatibility after structure size changes. Since most implementations will define **uid_t** as a short, the only substantive change will be a reduction in the size of the **passwd** structure. Consequently, implementations with an overriding concern for object compatibility can pad the structure back to its current size. For that reason, this problem was not considered critical enough to warrant the addition of a separate type to POSIX.1.

The types **uid_t** and **gid_t** are magic cookies. There is no {UID_MAX} defined by POSIX.1, and no structure imposed on **uid_t** and **gid_t** other than that they be positive arithmetic types. (In fact, they could be **unsigned char**.) There is no maximum or minimum specified for the number of distinct user or group IDs.

B.3 System Interfaces

See the RATIONALE sections on the individual reference pages.

B.3.1 Examples for Spawn

The following long examples are provided in the Rationale (Informative) volume of IEEE Std. 1003.1-200x as a supplement to the reference page for *spawn*( ).

Example Library Implementation of Spawn

The *posix_spawn*( ) or *posix_spawnp*( ) functions provide the following:

- Simply start a process executing a process image. This is the simplest application for process creation, and it may cover most executions of POSIX *fork*( ).
- Support I/O redirection, including pipes.
- Run the child under a user and group ID in the domain of the parent.
- Run the child at any priority in the domain of the parent.

The *posix_spawn*( ) or *posix_spawnp*( ) functions do not cover every possible use of the *fork*( ) function, but they do span the common applications: typical use by a shell and a login utility.

The price for an application is that before it calls *posix_spawn*( ) or *posix_spawnp*( ), the parent must adjust to a state that *posix_spawn*( ) or *posix_spawnp*( ) can map to the desired state for the child. Environment changes require the parent to save some of its state and restore it afterwards. The effective behavior of a successful invocation of *posix_spawn*( ) is as if the operation were implemented with POSIX operations as follows:

```
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sched.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>
#include <string.h>
#include <signal.h>

/* #include <spawn.h>*/
/*******************************************/
/* Things that could be defined in spawn.h */
/*******************************************/
typedef struct
    {
    short posix_attr_flags;
#define POSIX_SPAWN_SETPGROUP       0x1
#define POSIX_SPAWN_SETSIGMASK      0x2
#define POSIX_SPAWN_SETSIGDEF       0x4
#define POSIX_SPAWN_SETSCHEDULER    0x8
#define POSIX_SPAWN_SETSCHEDPARAM   0x10
#define POSIX_SPAWN_RESETIDS        0x20
    pid_t posix_attr_pgroup;
    sigset_t posix_attr_sigmask;
    sigset_t posix_attr_sigdefault;
```

```
    int posix_attr_schedpolicy;
    struct sched_param posix_attr_schedparam;
    } posix_spawnattr_t;

typedef char *posix_spawn_file_actions_t;

int posix_spawn_file_actions_init(
        posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_destroy(
        posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_addclose(
        posix_spawn_file_actions_t *file_actions, int fildes);
int posix_spawn_file_actions_adddup2(
        posix_spawn_file_actions_t *file_actions, int fildes,
        int newfildes);
int posix_spawn_file_actions_addopen(
        posix_spawn_file_actions_t *file_actions, int fildes,
        const char *path, int oflag, mode_t mode);
int posix_spawnattr_init(posix_spawnattr_t *attr);
int posix_spawnattr_destroy(posix_spawnattr_t *attr);
int posix_spawnattr_getflags(const posix_spawnattr_t *attr, short *lags);
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
int posix_spawnattr_getpgroup(const posix_spawnattr_t *attr,
        pid_t *pgroup);
int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *attr,
        int *schedpolicy);
int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
        int schedpolicy);
int posix_spawnattr_getschedparam(const posix_spawnattr_t *attr,
        struct sched_param *schedparam);
int posix_spawnattr_setschedparam(posix_spawnattr_t *attr,
        const struct sched_param *schedparam);
int posix_spawnattr_getsigmask(const posix_spawnattr_t *attr,
        sigset_t *sigmask);
int posix_spawnattr_setsigmask(posix_spawnattr_t *attr,
        const sigset_t *sigmask);
int posix_spawnattr_getdefault(const posix_spawnattr_t *attr,
        sigset_t *sigdefault);
int posix_spawnattr_setsigdefault(posix_spawnattr_t *attr,
        const sigset_t *sigdefault);
int posix_spawn(pid_t *pid, const char *path,
        const posix_spawn_file_actions_t *file_actions,
        const posix_spawnattr_t *attrp, char * const argv[],
        char * const envp[]);
int posix_spawnp(pid_t *pid, const char *file,
        const posix_spawn_file_actions_t *file_actions,
        const posix_spawnattr_t *attrp, char * const argv[],
        char * const envp[]);

/*****************************************/
/* Example posix_spawn() library routine */
/*****************************************/
int posix_spawn(pid_t *pid,
```

```
 const char *path,
 const posix_spawn_file_actions_t *file_actions,
 const posix_spawnattr_t *attrp,
 char * const argv[],
 char * const envp[])
{
/* Create process */
if((*pid=fork()) == (pid_t)0)
    {
    /* This is the child process */
    /* Worry about process group */
    if(attrp->posix_attr_flags & POSIX_SPAWN_SETPGROUP)
        {
        /* Override inherited process group */
        if(setpgid(0, attrp->posix_attr_pgroup) != 0)
            {
            /* Failed */
            exit(127);
            }
        }
     /* Worry about process signal mask */
     if(attrp->posix_attr_flags & POSIX_SPAWN_SETSIGMASK)
         {
         /* Set the signal mask (can't fail) */
         sigprocmask(SIG_SETMASK , &attrp->posix_attr_sigmask,
             NULL);
         }
     /* Worry about resetting effective user and group IDs */
     if(attrp->posix_attr_flags & POSIX_SPAWN_RESETIDS)
         {
         /* None of these can fail for this case. */
         setuid(getuid());
         setgid(getgid());
         }
     /* Worry about defaulted signals */
     if(attrp->posix_attr_flags & POSIX_SPAWN_SETSIGDEF)
         {
         struct sigaction deflt;
         sigset_t all_signals;
         int s;
         /* Construct default signal action */
         deflt.sa_handler = SIG_DFL;
         deflt.sa_flags = 0;
         /* Construct the set of all signals */
         sigfillset(&all_signals);
         /* Loop for all signals */
         for(s=0; sigismember(&all_signals,s); s++)
             {
             /* Signal to be defaulted? */
```

```
        if(sigismember(&attrp->posix_attr_sigdefault,s))
            {
            /* Yes; default this signal */
            if(sigaction(s, &deflt, NULL) == -1)
                {
                /* Failed */
                exit(127);
                }
            }
        }
    }
/* Worry about the fds if we are to map them */
if(file_actions != NULL)
    {
    /* Loop for all actions in object file_actions */
    /*(implementation dives beneath abstraction)*/
    char *p = *file_actions;
    while(*p != ’ ’)
        {
        if(strncmp(p,"close(",6) == 0)
            {
            int fd;
            if(sscanf(p+6,"%d)",&fd) != 1)
                {
                exit(127);
                }
            if(close(fd) == -1) exit(127);
            }
        else if(strncmp(p,"dup2(",5) == 0)
            {
            int fd,newfd;
            if(sscanf(p+5,"%d,%d)",&fd,&newfd) != 2)
                {
                exit(127);
                }
            if(dup2(fd, newfd) == -1) exit(127);
            }
        else if(strncmp(p,"open(",5) == 0)
            {
            int fd,oflag;
            mode_t mode;
            int tempfd;
            char path[1000]; /* Should be dynamic */
            char *q;
            if(sscanf(p+5,"%d,",&fd) != 1)
                {
                 exit(127);
                }
            p = strchr(p, ’,’) + 1;
            q = strchr(p, ’*’);
            if(q == NULL) exit(127);
            strncpy(path, p, q-p);
```

```
                    path[q-p] = ' ';
                if(sscanf(q+1,"%o,%o)",&oflag,&mode)!=2)
                    {
                    exit(127);
                    }
                if(close(fd) == -1)
                    {
                    if(errno != EBADF) exit(127);
                    }
                tempfd = open(path, oflag, mode);
                if(tempfd == -1) exit(127);
                if(tempfd != fd)
                    {
                    if(dup2(tempfd,fd) == -1)
                        {
                        exit(127);
                        }
                    if(close(tempfd) == -1)
                        {
                        exit(127);
                        }
                    }
                }
                else
                {
                exit(127);
                }
            p = strchr(p, ')') + 1;
            }
        }
        /* Worry about setting new scheduling policy and parameters */
        if(attrp->posix_attr_flags & POSIX_SPAWN_SETSCHEDULER)
            {
            if(sched_setscheduler(0, attrp->posix_attr_schedpolicy,
                &attrp->posix_attr_schedparam) == -1)
                {
                exit(127);
                }
            }
        /* Worry about setting only new scheduling parameters */
        if(attrp->posix_attr_flags & POSIX_SPAWN_SETSCHEDPARAM)
            {
            if(sched_setparam(0, &attrp->posix_attr_schedparam)==-1)
                {
                exit(127);
                }
            }
        /* Now execute the program at path */
        /* Any fd that still has FD_CLOEXEC set will be closed */
        execve(path, argv, envp);
        exit(127); /* exec failed */
```

```
            }
            else
            {
            /* This is the parent (calling) process */
            if(*pid == (pid_t)-1) return errno;
            return 0;
            }
        }
    /*******************************************************/
    /* Here is a crude but effective implementation of the */
    /* file action object operators which store actions as */
    /* concatenated token separated strings.               */
    /*******************************************************/
    /* Create object with no actions. */
    int posix_spawn_file_actions_init(
            posix_spawn_file_actions_t *file_actions)
        {
        *file_actions = malloc(sizeof(char));
        if(*file_actions == NULL) return ENOMEM;
        strcpy(*file_actions, "");
        return 0;
        }
    /* Free object storage and make invalid. */
    int posix_spawn_file_actions_destroy(
            posix_spawn_file_actions_t *file_actions)
        {
        free(*file_actions);
        *file_actions = NULL;
        return 0;
        }
    /* Add a new action string to object. */
    static int add_to_file_actions(
            posix_spawn_file_actions_t *file_actions,
                char *new_action)
        {
        *file_actions = realloc
            (*file_actions, strlen(*file_actions)+strlen(new_action)+1);
        if(*file_actions == NULL) return ENOMEM;
        strcat(*file_actions, new_action);
        return 0;
        }
    /* Add a close action to object. */
    int posix_spawn_file_actions_addclose(
            posix_spawn_file_actions_t *file_actions, int fildes)
        {
        char temp[100];
        sprintf(temp, "close(%d)", fildes);
        return add_to_file_actions(file_actions, temp);
        }
```

```
/* Add a dup2 action to object. */
int posix_spawn_file_actions_adddup2(
        posix_spawn_file_actions_t *file_actions, int fildes,
        int newfildes)
    {
    char temp[100];
    sprintf(temp, "dup2(%d,%d)", fildes, newfildes);
    return add_to_file_actions(file_actions, temp);
    }

/* Add an open action to object. */
int posix_spawn_file_actions_addopen(
        posix_spawn_file_actions_t *file_actions, int fildes,
        const char *path, int oflag, mode_t mode)
    {
    char temp[100];
    sprintf(temp, "open(%d,%s*%o,%o)", fildes, path, oflag, mode);
    return add_to_file_actions(file_actions, temp);
    }

/*******************************************************/
/* Here is a crude but effective implementation of the */
/* spawn attributes object functions which manipulate  */
/* the individual attributes.                          */
/*******************************************************/
/* Initialize object with default values. */
int posix_spawnattr_init(
        posix_spawnattr_t *attr)
    {
    attr->posix_attr_flags=0;
    attr->posix_attr_pgroup=0;
    /* Default value of signal mask is the parent’s signal mask; */
    /* other values are also allowed */
    sigprocmask(0,NULL,&attr->posix_attr_sigmask);
    sigemptyset(&attr->posix_attr_sigdefault);
    /* Default values of scheduling attr inherited from the parent; */
    /* other values are also allowed */
    attr->posix_attr_schedpolicy=sched_getscheduler(0);
    sched_getparam(0,&attr->posix_attr_schedparam);
    return 0;
    }

int posix_spawnattr_destroy(posix_spawnattr_t *attr)
    {
    /* No action needed */
    return 0;
    }

int posix_spawnattr_getflags(const posix_spawnattr_t *attr,
        short *flags)
    {
    *flags=attr->posix_attr_flags;
    return 0;
    }
```

```
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags)
    {
    attr->posix_attr_flags=flags;
    return 0;
    }

int posix_spawnattr_getpgroup(const posix_spawnattr_t *attr,
        pid_t *pgroup)
    {
    *pgroup=attr->posix_attr_pgroup;
    return 0;
    }

int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup)
    {
    attr->posix_attr_pgroup=pgroup;
    return 0;
    }

int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *attr,
        int *schedpolicy)
    {
    *schedpolicy=attr->posix_attr_schedpolicy;
    return 0;
    }

int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
        int schedpolicy)
    {
    attr->posix_attr_schedpolicy=schedpolicy;
    return 0;
    }

int posix_spawnattr_getschedparam(const posix_spawnattr_t *attr,
        struct sched_param *schedparam)
    {
    *schedparam=attr->posix_attr_schedparam;
    return 0;
    }

int posix_spawnattr_setschedparam(posix_spawnattr_t *attr,
        const struct sched_param *schedparam)
    {
    attr->posix_attr_schedparam=*schedparam;
    return 0;
    }

int posix_spawnattr_getsigmask(const posix_spawnattr_t *attr,
        sigset_t *sigmask)
    {
    *sigmask=attr->posix_attr_sigmask;
    return 0;
    }

int posix_spawnattr_setsigmask(posix_spawnattr_t *attr,
        const sigset_t *sigmask)
```

```
    {
    attr->posix_attr_sigmask=*sigmask;
    return 0;
    }
int posix_spawnattr_getsigdefault(const posix_spawnattr_t *attr,
        sigset_t *sigdefault)
    {
    *sigdefault=attr->posix_attr_sigdefault;
    return 0;
    }
int posix_spawnattr_setsigdefault(posix_spawnattr_t *attr,
        const sigset_t *sigdefault)
    {
    attr->posix_attr_sigdefault=*sigdefault;
    return 0;
    }
```

I/O Redirection with Spawn

I/O redirection with *posix_spawn*( ) or *posix_spawnp*( ) is accomplished by crafting a *file_actions* argument to effect the desired redirection. Such a redirection follows the general outline of the following example:

```
/* To redirect new standard output (fd 1) to a file, */
/* and redirect new standard input (fd 0) from my fd socket_pair[1], */
/* and close my fd socket_pair[0] in the new process. */
posix_spawn_file_actions_t file_actions;
posix_spawn_file_actions_init(&file_actions);
posix_spawn_file_actions_addopen(&file_actions, 1, "newout", ...);
posix_spawn_file_actions_dup2(&file_actions, socket_pair[1], 0);
posix_spawn_file_actions_close(&file_actions, socket_pair[0]);
posix_spawn_file_actions_close(&file_actions, socket_pair[1]);
posix_spawn(..., &file_actions, ...);
posix_spawn_file_actions_destroy(&file_actions);
```

Spawning a Process Under a New User ID

Spawning a process under a new user ID follows the outline shown in the following example:

```
Save = getuid();
setuid(newid);
posix_spawn(...);
setuid(Save);
```

Rationale (Informative)

Part C:

Shell and Utilities

The Open Group

Appendix C

Rationale for Shell and Utilities

C.1 Introduction

C.1.1 Scope

Refer to Section A.1.1 (on page 3311).

C.1.2 Conformance

Refer to Section A.2 (on page 3317).

C.1.3 Normative References

There is no additional rationale provided for this section.

C.1.4 Changes from Issue 4

The change history is provided as an informative section, to track changes from previous issues of IEEE Std. 1003.1-200x that comprised earlier versions of the Single UNIX Specification.

C.1.4.1 Changes from Issue 4 to Issue 4, Version 2

There is no additional rationale provided for this section.

C.1.4.2 Changes from Issue 4, Version 2 to Issue 5

There is no additional rationale provided for this section.

C.1.4.3 Changes from Issue 5 to Issue 6

There is no additional rationale provided for this section.

C.1.5 Terminology

Refer to Section A.1.4 (on page 3313).

C.1.6 Definitions

Refer to Section A.3 (on page 3321).

C.1.7 Relationship to Other Documents

C.1.7.1 System Interfaces

It has been pointed out that the Shell and Utilities volume of IEEE Std. 1003.1-200x assumes that a great deal of functionality from the System Interfaces volume of IEEE Std. 1003.1-200x is present, but never states exactly how much (and strictly does not need to since both are mandated on a conforming system). This section is an attempt to clarify the assumptions.

C.1.8 Portability

Refer to Section A.1.5 (on page 3315).

C.1.8.1 Codes

Refer to Section A.1.5.1 (on page 3315).

C.1.9 Utility Limits

This section grew out of an idea that originated with the original POSIX.1, in the tables of system limits for the *sysconf*( ) and *pathconf*( ) functions. The idea being that a conforming application can be written to use the most restrictive values that a minimal system can provide, but it should not have to. The values provided represent compromises so that some vendors can use historically limited versions of UNIX system utilities. They are the highest values that a strictly conforming application can assume, given no other information.

However, by using the *getconf* utility or the *sysconf*( ) function, the elegant application can be tailored to more liberal values on some of the specific instances of specific implementations.

There is no explicitly stated requirement that an implementation provide finite limits for any of these numeric values; the implementation is free to provide essentially unbounded capabilities (where it makes sense), stopping only at reasonable points such as {ULONG_MAX} (from the ISO C standard). Therefore, applications desiring to tailor themselves to the values on a particular implementation need to be ready for possibly huge values; it may not be a good idea to allocate blindly a buffer for an input line based on the value of {LINE_MAX}, for instance. However, unlike the System Interfaces volume of IEEE Std. 1003.1-200x, there is no set of limits that return a special indication meaning ''unbounded''. The implementation should always return an actual number, even if the number is very large.

The statement:

> ''It is not guaranteed that the application ...''

is an indication that many of these limits are designed to ensure that implementors design their utilities without arbitrary constraints related to unimaginative programming. There are certainly conditions under which combinations of options can cause failures that would not render an implementation non-conforming. For example, {EXPR_NEST_MAX} and {ARG_MAX} could collide when expressions are large; combinations of {BC_SCALE_MAX} and {BC_DIM_MAX} could exceed virtual memory.

In the Shell and Utilities volume of IEEE Std. 1003.1-200x, the notion of a limit being guaranteed for the process lifetime, as it is in the System Interfaces volume of IEEE Std. 1003.1-200x, is not as useful to a shell script. The *getconf* utility is probably a process itself, so the guarantee would be without value. Therefore, the Shell and Utilities volume of IEEE Std. 1003.1-200x requires the guarantee to be for the session lifetime. This will mean that many vendors will either return very conservative values or possibly implement *getconf* as a built-in.

It may seem confusing to have limits that apply only to a single utility grouped into one global section. However, the alternative, which would be to disperse them out into their utility description sections, would cause great difficulty when *sysconf*( ) and *getconf* were described. Therefore, the standard developers chose the global approach.

Each language binding could provide symbol names that are slightly different than are shown here. For example, the C-Language Binding option adds a leading underscore to the symbols as a prefix.

The following comments describe selection criteria for the symbols and their values:

{ARG_MAX}

This is defined by the System Interfaces volume of IEEE Std. 1003.1-200x. Unfortunately, it is very difficult for a portable application to deal with this value, as it does not know how much of its argument space is being consumed by the environment variables of the user.

{BC_BASE_MAX}
{BC_DIM_MAX}
{BC_SCALE_MAX}

These were originally one value, {BC_SCALE_MAX}, but it was unreasonable to link all three concepts into one limit.

{CHILD_MAX}

This is defined by the System Interfaces volume of IEEE Std. 1003.1-200x.

{COLL_WEIGHTS_MAX}

The weights assigned to **order** can be considered as ''passes'' through the collation algorithm.

{EXPR_NEST_MAX}

The value for expression nesting was borrowed from the ISO C standard.

{LINE_MAX}

This is a global limit that affects all utilities, unless otherwise noted. The {MAX_CANON} value from the System Interfaces volume of IEEE Std. 1003.1-200x may further limit input lines from terminals. The {LINE_MAX} value was the subject of much debate and is a compromise between those who wished to have unlimited lines and those who understood that many historical utilities were written with fixed buffers. Frequently, utility writers selected the UNIX system constant {BUFSIZ} to allocate these buffers; therefore, some utilities were limited to 512 bytes for I/O lines, while others achieved 4 096 bytes or greater.

It should be noted that {LINE_MAX} applies only to input line length; there is no requirement in IEEE Std. 1003.1-200x that limits the length of output lines. Utilities such as *awk*, *sed*, and *paste* could theoretically construct lines longer than any of the input lines they received, depending on the options used or the instructions from the application. They are not required to truncate their output to {LINE_MAX}. It is the responsibility of the application to deal with this. If the output of one of those utilities is to be piped into another of the standard utilities, line length restrictions will have to be considered; the *fold* utility, among others, could be used to ensure that only reasonable line lengths reach utilities or applications.

{LINK_MAX}

This is defined by the System Interfaces volume of IEEE Std. 1003.1-200x.

{MAX_CANON}
{MAX_INPUT}
{NAME_MAX}
{NGROUPS_MAX}
{OPEN_MAX}
{PATH_MAX}
{PIPE_BUF}

These limits are defined by the System Interfaces volume of IEEE Std. 1003.1-200x. Note that the byte lengths described by some of these values continue to represent bytes, even if the applicable character set uses a multi-byte encoding.

{RE_DUP_MAX}

The value selected is consistent with historical practice. Although the name implies that it applies to all REs, only BREs use the interval notation **\{***m,n***\}** addressed by this limit.

{POSIX2_SYMLINKS}

The {POSIX2_SYMLINKS} variable indicates that the underlying operating system supports the creation of symbolic links in specific directories. Many of the utilities defined in IEEE Std. 1003.1-200x that deal with symbolic links do not depend on this value. For example, a utility that follows symbolic links (or does not, as the case may be) will only be affected by a symbolic link if it encounters one. Presumably, a file system that does not support symbolic links will not contain any. This variable does affect such utilities as *ln* **−s** and *pax* that attempt to create symbolic links.

{POSIX2_SYMLINKS} was developed even though there is no comparable configuration value in the IEEE P1003.1a draft standard.

There are different limits associated with command lines and input to utilities, depending on the method of invocation. In the case of a C program *exec*-ing a utility, {ARG_MAX} is the underlying limit. In the case of the shell reading a script and *exec*-ing a utility, {LINE_MAX} limits the length of lines the shell is required to process, and {ARG_MAX} will still be a limit. If a user is entering a command on a terminal to the shell, requesting that it invoke the utility, {MAX_INPUT} may restrict the length of the line that can be given to the shell to a value below {LINE_MAX}.

When an option is supported, *getconf* returns a value of 1. For example, when C development is supported:

```
if [ "$(getconf POSIX2_C_DEV)" −eq 1 ]; then
    echo C supported
fi
```

The *sysconf*( ) function in the C-Language Binding option would return 1.

The following comments describe selection criteria for the symbols and their values:

POSIX2_C_BIND
POSIX2_C_DEV
POSIX2_FORT_DEV
POSIX2_FORT_RUN
POSIX2_SW_DEV
POSIX2_UPE

It is possible for some (usually privileged) operations to remove utilities that support these options or otherwise to render these options unsupported. The header files, the *sysconf*( ) function, or the *getconf* utility will not necessarily detect such actions, in which case they should not be considered as rendering the implementation non-conforming. A test suite should not attempt tests such as:

```
rm /usr/bin/c89
getconf POSIX2_C_DEV
```

POSIX2_LOCALEDEF

This symbol was introduced to allow implementations to restrict supported locales to only those supplied by the implementation.

C.1.10 Grammar Conventions

There is no additional rationale for this section.

C.1.11 Utility Description Defaults

This section is arranged with headings in the same order as all the utility descriptions. It is a collection of related and unrelated information concerning

1. The default actions of utilities
2. The meanings of notations used in IEEE Std. 1003.1-200x that are specific to individual utility sections

Although this material may seem out of place here, it is important that this information appear before any of the utilities to be described later.

NAME

There is no additional rationale provided for this section.

SYNOPSIS

There is no additional rationale provided for this section.

DESCRIPTION

There is no additional rationale provided for this section.

OPTIONS

Although it has not always been possible, the standard developers tried to avoid repeating information to reduce the risk that duplicate explanations could each be modified differently.

The need to recognize -- is required because portable applications need to shield their operands from any arbitrary options that the implementation may provide as an extension. For example, if the standard utility *foo* is listed as taking no options, and the application needed to give it a path name with a leading hyphen, it could safely do it as:

```
foo -- -myfile
```

and avoid any problems with **-m** used as an extension.

OPERANDS

The usage of - is never shown in the SYNOPSIS. Similarly, the usage of -- is never shown.

The requirement for processing operands in command-line order is to avoid a ''WeirdNIX'' utility that might choose to sort the input files alphabetically, by size, or by directory order. Although this might be acceptable for some utilities, in general the programmer has a right to know exactly what order will be chosen.

Some of the standard utilities take multiple *file* operands and act as if they were processing the concatenation of those files. For example:

```
asa file1 file2
```

and:

```
cat file1 file2 | asa
```

have similar results when questions of file access, errors, and performance are ignored. Other utilities such as *grep* or *wc* have completely different results in these two cases. This latter type of utility is always identified in its DESCRIPTION or OPERANDS sections, whereas the former is not. Although it might be possible to create a general assertion about the former case, the following points must be addressed:

- Access times for the files might be different in the operand case *versus* the *cat* case.
- The utility may have error messages that are cognizant of the input file name, and this added value should not be suppressed. (As an example, *awk* sets a variable with the file name at each file boundary.)

STDIN

There is no additional rationale provided for this section.

INPUT FILES

A conforming application cannot assume the following three commands are equivalent:

```
tail -n +2 file
(sed -n 1q; cat) < file
cat file | (sed -n 1q; cat)
```

The second command is equivalent to the first only when the file is seekable. In the third command, if the file offset in the open file description were not unspecified, *sed* would have to be implemented so that it read from the pipe 1 byte at a time or it would have to employ some method to seek backwards on the pipe. Such functionality is not defined currently in POSIX.1 and does not exist on all historical systems. Other utilities, such as *head*, *read*, and *sh*, have similar properties, so the restriction is described globally in this section.

The definition of *text file* is strictly enforced for input to the standard utilities; very few of them list exceptions to the undefined results called for here. (Of course, ‘‘undefined’’ here does not mean that historical implementations necessarily have to change to start indicating error conditions. Conforming applications cannot rely on implementations succeeding or failing when non-text files are used.)

The utilities that allow line continuation are generally those that accept input languages, rather than pure data. It would be unusual for an input line of this type to exceed {LINE_MAX} bytes and unreasonable to require that the implementation allow unlimited accumulation of multiple lines, each of which could reach {LINE_MAX}. Thus, for a portable application the total of all the continued lines in a set cannot exceed {LINE_MAX}.

The format description is intended to be sufficiently rigorous to allow other applications to generate these input files. However, since <blank>s can legitimately be included in some of the fields described by the standard utilities, particularly in locales other than the POSIX locale, this intent is not always realized.

ENVIRONMENT VARIABLES

There is no additional rationale provided for this section.

ASYNCHRONOUS EVENTS

Because there is no language prohibiting it, a utility is permitted to catch a signal, perform some additional processing (such as deleting temporary files), restore the default signal action (or action inherited from the parent process), and resignal itself.

STDOUT

The format description is intended to be sufficiently rigorous to allow post-processing of output by other programs, particularly by an *awk* or *lex* parser.

STDERR

This section does not describe error messages that refer to incorrect operation of the utility. Consider a utility that processes program source code as its input. This section is used to describe messages produced by a correctly operating utility that encounters an error in the program source code on which it is processing. However, a message indicating that the utility had insufficient memory in which to operate would not be described.

Some utilities have traditionally produced warning messages without returning a non-zero exit status; these are specifically noted in their sections. Other utilities shall not write to standard error if they complete successfully, unless the implementation provides some sort of extension to increase the verbosity or debugging level.

The format descriptions are intended to be sufficiently rigorous to allow post-processing of output by other programs.

OUTPUT FILES

The format description is intended to be sufficiently rigorous to allow post-processing of output by other programs, particularly by an *awk* or *lex* parser.

Receipt of the SIGQUIT signal should generally cause termination (unless in some debugging mode) that would bypass any attempted recovery actions.

EXTENDED DESCRIPTION

There is no additional rationale provided for this section.

EXIT STATUS

Note the additional discussion of exit values in *Exit Status for Commands* in the *sh* utility. It describes requirements for returning exit values greater than 125.

A utility may list zero as a successful return, 1 as a failure for a specific reason, and greater than 1 as ''an error occurred''. In this case, unspecified conditions may cause a 2 or 3, or other value, to be returned. A strictly conforming application should be written so that it tests for successful exit status values (zero in this case), rather than relying upon the single specific error value listed in IEEE Std. 1003.1-200x. In that way, it will have maximum portability, even on implementations with extensions.

The standard developers are aware that the general non-enumeration of errors makes it difficult to write test suites that test the *incorrect* operation of utilities. There are some historical implementations that have expended effort to provide detailed status messages and a helpful environment to bypass or explain errors, such as prompting, retrying, or ignoring unimportant syntax errors; other implementations have not. Since there is no realistic way to mandate system behavior in cases of undefined application actions or system problems—in a manner acceptable to all cultures and environments—attention has been limited to the correct operation of utilities by the conforming application. Furthermore, the portable application does not need detailed information concerning errors that it caused through incorrect usage or that it cannot correct.

There is no description of defaults for this section because all of the standard utilities specify something (or explicitly state ''Unspecified'') for exit status.

CONSEQUENCES OF ERRORS

Several actions are possible when a utility encounters an error condition, depending on the severity of the error and the state of the utility. Included in the possible actions of various utilities are: deletion of temporary or intermediate work files; deletion of incomplete files; and validity checking of the file system or directory.

The text about recursive traversing is meant to ensure that utilities such as *find* process as many files in the hierarchy as they can. They should not abandon all of the hierarchy at the first error and resume with the next command-line operand, but should attempt to keep going.

APPLICATION USAGE

This section provides additional caveats, issues, and recommendations to the developer.

EXAMPLES

This section provides sample usage.

RATIONALE

There is no additional rationale provided for this section.

FUTURE DIRECTIONS

FUTURE DIRECTIONS sections act as pointers to related work that may impact the interface in the future, and often cautions the developer to architect the code to account for a change in this area. Note that a future directions statement should not be taken as a commitment to adopt a feature or interface in the future.

SEE ALSO

There is no additional rationale provided for this section.

CHANGE HISTORY

There is no additional rationale provided for this section.

C.1.12 Considerations for Utilities in Support of Files of Arbitrary Size

This section is intended to clarify the requirements for utilities in support of large files.

The utilities listed in this section are utilities which are used to perform administrative tasks such as to create, move, copy, remove, change the permissions, or measure the resources of a file. They are useful both as end-user tools and as utilities invoked by applications during software installation and operation.

The *chgrp*, *chmod*, *chown*, *ln*, and *rm* utilities probably require use of large file capable versions of *stat*( ), *lstat*( ), *ftw*( ), and the **stat** structure.

The *cat*, *cksum*, *cmp*, *cp*, *dd*, *mv*, *sum*, and *touch* utilities probably require use of large file capable versions of *creat*( ), *open*( ), and *fopen*( ).

The *cat*, *cksum*, *cmp*, *dd*, *df*, *du*, *ls*, and *sum* utilities may require writing large integer values. For example:

- The *cat* utility might have a **–n** option which counts <newline>s.
- The *cksum* and *ls* utilities report file sizes.
- The *cmp* utility reports the line number at which the first difference occurs, and also has a **–l** option which reports file offsets.
- The *dd*, *df*, *du*, *ls*, and *sum* utilities report block counts.

The *dd*, *find*, and *test* utilities may need to interpret command arguments that contain 64-bit values. For *dd*, the arguments include *skip*=*n*, *seek*=*n*, and *count*=*n*. For *find*, the arguments include **–size***n*. For *test*, the arguments are those associated with algebraic comparisons.

The *df* utility might need to access large file systems with *statvfs*( ).

The *ulimit* utility will need to use large file capable versions of *getrlimit*( ) and *setrlimit*( ) and be able to read and write large integer values.

C.2 Shell Command Language

C.2.1 Shell Introduction

The System V shell was selected as the starting point for the Shell and Utilities volume of IEEE Std. 1003.1-200x. The BSD C shell was excluded from consideration for the following reasons:

- Most historically portable shell scripts assume the Version 7 Bourne shell, from which the System V shell is derived.
- The majority of tutorial materials on shell programming assume the System V shell.

The construct `"#!"` is reserved for implementations wishing to provide that extension. If it were not reserved, the Shell and Utilities volume of IEEE Std. 1003.1-200x would disallow it by forcing it to be a comment. As it stands, a POSIX-conforming application must not use `"#!"` as the first two characters of the file. An XSI-conforming application can use the construct `"#!"`, since on XSI-conformant systems this is defined to denote an executable script, which matches historical practice. Invention of new meanings or extensions to the `"#!"` construct were rejected since they are beyond the scope of IEEE Std. 1003.1-200x.

C.2.2 Quoting

There is no additional rationale for this section.

C.2.2.1 Escape Character (Backslash)

There is no additional rationale for this section.

C.2.2.2 Single-Quotes

A backslash cannot be used to escape a single-quote in a single-quoted string. An embedded quote can be created by writing, for example: `"'a'\''b'"`, which yields `"a'b"`. (See the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.6.5, Field Splitting for a better understanding of how portions of words are either split into fields or remain concatenated.) A single token can be made up of concatenated partial strings containing all three kinds of quoting or escaping, thus permitting any combination of characters.

C.2.2.3 Double-Quotes

The escaped <newline> used for line continuation is removed entirely from the input and is not replaced by any white space. Therefore, it cannot serve as a token separator.

In double-quoting, if a backslash is immediately followed by a character that would be interpreted as having a special meaning, the backslash is deleted and the subsequent character is taken literally. If a backslash does not precede a character that would have a special meaning, it is left in place unmodified and the character immediately following it is also left unmodified. Thus, for example:

`"\$"` → `$`

`"\a"` → `\a`

It would be desirable to include the statement "The characters from an enclosed `"${"` to the matching `'}'` shall not be affected by the double quotes", similar to the one for `"$()"`. However, historical practice in the System V shell prevents this.

The requirement that double-quotes be matched inside `"${...}"` within double-quotes and the rule for finding the matching `'}'` in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.6.2, Parameter Expansion eliminate several subtle inconsistencies in expansion for historical shells in rare cases; for example:

```
"${foo-bar"}
```

yields **bar** when **foo** is not defined, and is an invalid substitution when **foo** is defined, in many historical shells. The differences in processing the `"${...}"` form have led to inconsistencies between historical systems. A consequence of this rule is that single-quotes cannot be used to quote the `'}'` within `"${...}"`; for example:

```
unset bar
foo="${bar-'}'}"
```

is invalid because the `"${...}"` substitution contains an unpaired unescaped single-quote. The backslash can be used to escape the `'}'` in this example to achieve the desired result:

```
unset bar
foo="${bar-\}}"
```

The differences in processing the `"${...}"` form have led to inconsistencies between the historical System V shell, BSD, and KornShells, and the text in the Shell and Utilities volume of IEEE Std. 1003.1-200x is an attempt to converge them without breaking too many applications. The only alternative to this compromise between shells would be to make the behavior unspecified whenever the literal characters `'''`, `'{'`, `'}'`, and `'"'` appear within `"${...}"`. To write a portable script that uses these values, a user would have to assign variables; for example:

```
squote=\' dquote=\" lbrace='{' rbrace='}'
${foo-$squote$rbrace$squote}
```

rather than:

```
${foo-"'}'"}
```

Some systems have allowed the end of the word to terminate the backquoted command substitution, such as in:

```
"`echo hello"
```

This usage is undefined; the matching backquote is required by the Shell and Utilities volume of IEEE Std. 1003.1-200x. The other undefined usage can be illustrated by the example:

```
sh -c '` echo "foo`'
```

The description of the recursive actions involving command substitution can be illustrated with an example. Upon recognizing the introduction of command substitution, the shell parses input (in a new context), gathering the source for the command substitution until an unbalanced `')'` or `'''` is located. For example, in the following:

```
echo "$(date; echo "
    one" )"
```

the double-quote following the *echo* does not terminate the first double-quote; it is part of the command substitution script. Similarly, in:

```
echo "$(echo *)"
```

the asterisk is not quoted since it is inside command substitution; however:

```
echo "$(echo "*")"
```

is quoted (and represents the asterisk character itself).

C.2.3 Token Recognition

The `"(("` and `"))"` symbols are control operators in the KornShell, used for an alternative syntax of an arithmetic expression command. A portable application cannot use `"(("` as a single token (with the exception of the `"$(("` form for shell arithmetic).

The (3) rule about combining characters to form operators is not meant to preclude systems from extending the shell language when characters are combined in otherwise invalid ways. Portable applications cannot use invalid combinations, and test suites should not penalize systems that take advantage of this fact. For example, the unquoted combination `"|&"` is not valid in a POSIX script, but has a specific KornShell meaning.

The (10) rule about `'#'` as the current character is the first in the sequence in which a new token is being assembled. The `'#'` starts a comment only when it is at the beginning of a token. This rule is also written to indicate that the search for the end-of-comment does not consider escaped <newline> specially, so that a comment cannot be continued to the next line.

C.2.3.1 Alias Substitution

The alias capability was added in the UPE because it is widely used in historical implementations by interactive users.

The definition of *alias name* precludes an alias name containing a slash character. Since the text applies to the command words of simple commands, reserved words (in their proper places) cannot be confused with aliases.

The placement of alias substitution in token recognition makes it clear that it precedes all of the word expansion steps.

An example concerning trailing <blank> characters and reserved words follows. If the user types:

```
$ alias foo="/bin/ls "
$ alias while="/"
```

The effect of executing:

```
$ while true
> do
> echo "Hello, World"
> done
```

is a never-ending sequence of `"Hello, World"` strings to the screen. However, if the user types:

```
$ foo while
```

the result is an *ls* listing of **/**. Since the alias substitution for **foo** ends in a <space> character, the next word is checked for alias substitution. The next word, **while**, has also been aliased, so it is substituted as well. Since it is not in the proper position as a command word, it is not recognized as a reserved word.

If the user types:

```
$ foo; while
```

while retains its normal reserved-word properties.

C.2.4 Reserved Words

All reserved words are recognized syntactically as such in the contexts described. However, note that **in** is the only meaningful reserved word after a **case** or **for**; similarly, **in** is not meaningful as the first word of a simple command.

Reserved words are recognized only when they are delimited (that is, meet the definition of the Base Definitions volume of IEEE Std. 1003.1-200x, Section 3.437, Word), whereas operators are themselves delimiters. For instance, `'('` and `')'` are control operators, so that no <space> character is needed in (*list*). However, `'{'` and `'}'` are reserved words in { *list*;}, so that in this case the leading <space> character and semicolon are required.

The list of unspecified reserved words is from the KornShell, so portable applications cannot use them in places a reserved word would be recognized. This list contained **time** in early proposals, but it was removed when the *time* utility was selected for the Shell and Utilities volume of IEEE Std. 1003.1-200x.

There was a strong argument for promoting braces to operators (instead of reserved words), so they would be syntactically equivalent to subshell operators. Concerns about compatibility outweighed the advantages of this approach. Nevertheless, portable applications should consider quoting `'{'` and `'}'` when they represent themselves.

The restriction on ending a name with a colon is to allow future implementations that support named labels for flow control; see the RATIONALE for the *break* built-in utility .

It is possible that a future version of the Shell and Utilities volume of IEEE Std. 1003.1-200x may require that `'{'` and `'}'` be treated individually as control operators, although the token `"{}"` will probably be a special-case exemption from this because of the often-used *find*{} construct.

C.2.5 Parameters and Variables

C.2.5.1 Positional Parameters

There is no additional rationale for this section.

C.2.5.2 Special Parameters

Most historical implementations implement subshells by forking; thus, the special parameter `'$'` does not necessarily represent the process ID of the shell process executing the commands since the subshell execution environment preserves the value of `'$'`.

If a subshell were to execute a background command, the value of `"$!"` for the parent would not change. For example:

```
(
date &
echo $!
)
echo $!
```

would echo two different values for `"$!"`.

The `"$–"` special parameter can be used to save and restore *set* options:

```
Save=$(echo $- | sed 's/[ics]//g')
...
set +aCefnuvx
if [ -n "$Save" ]; then
    set -$Save
fi
```

The three options are removed using *sed* in the example because they may appear in the value of `"$-"` (from the *sh* command line), but are not valid options to *set*.

The descriptions of parameters `'*'` and `'@'` assume the reader is familiar with the field splitting discussion in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.6.5, Field Splitting and understands that portions of the word remain concatenated unless there is some reason to split them into separate fields.

Some examples of the `'*'` and `'@'` properties, including the concatenation aspects:

```
set "abc" "def ghi" "jkl"

echo $*    => "abc" "def" "ghi" "jkl"
echo "$*"  => "abc def ghi jkl"
echo $@    => "abc" "def" "ghi" "jkl"
```

but:

```
echo "$@"      => "abc" "def ghi" "jkl"
echo "xx$@yy"  => "xxabc" "def ghi" "jklyy"
echo "$@$@"    => "abc" "def ghi" "jklabc" "def ghi" "jkl"
```

In the preceding examples, the double-quote characters that appear after the `"=>"` do not appear in the output and are used only to illustrate word boundaries.

The following example illustrates the effect of setting *IFS* to a null string:

```
$ IFS=''
$ set foo bar bam
$ echo "$@"
foo bar bam
$ echo "$*"
foobarbam
$ unset IFS
$ echo "$*"
foo bar bam
```

C.2.5.3 *Shell Variables*

See the discussion of *IFS* in Section C.2.6.5 (on page 3532).

The prohibition on *LC_CTYPE* changes affecting lexical processing protects the shell implementor (and the shell programmer) from the ill effects of changing the definition of <blank> or the set of alphabetic characters in the current environment. It would probably not be feasible to write a compiled version of a shell script without this rule. The rule applies only to the current invocation of the shell and its subshells—invoking a shell script or performing *exec sh* would subject the new shell to the changes in *LC_CTYPE*.

Other common environment variables used by historical shells are not specified by the Shell and Utilities volume of IEEE Std. 1003.1-200x, but they should be reserved for the historical uses.

Tilde expansion for components of the *PATH* in an assignment such as:

```
PATH=~hlj/bin:~dwc/bin:$PATH
```

is a feature of some historical shells and is allowed by the wording of the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.6.1, Tilde Expansion. Note that the tildes are expanded during the assignment to *PATH*, not when *PATH* is accessed during command search.

The following entries represent additional information about variables included in the Shell and Utilities volume of IEEE Std. 1003.1-200x, or rationale for common variables in use by shells that have been excluded:

_ (Underscore.) While underscore is historical practice, its overloaded usage in the KornShell is confusing, and it has been omitted from the Shell and Utilities volume of IEEE Std. 1003.1-200x.

ENV This variable can be used to set aliases and other items local to the invocation of a shell. The file referred to by *ENV* differs from **$HOME/.profile** in that **.profile** is typically executed at session start-up, whereas the *ENV* file is executed at the beginning of each shell invocation. The *ENV* value is interpreted in a manner similar to a dot script, in that the commands are executed in the current environment and the file needs to be readable, but not executable. However, unlike dot scripts, no *PATH* searching is performed. This is used as a guard against Trojan Horse security breaches.

ERRNO This variable was omitted from the Shell and Utilities volume of IEEE Std. 1003.1-200x because the values of error numbers are not defined in IEEE Std. 1003.1-200x in a portable manner.

FCEDIT Since this variable affects only the *fc* utility, it has been omitted from this more global place. The value of *FCEDIT* does not affect the command line editing mode in the shell; see the description of *set* −**o** *vi* in the *set* built-in utility.

PS1 This variable is used for interactive prompts. Historically, the ‘‘superuser’’ has had a prompt of `'#'`. Since privileges are not required to be monolithic, it is difficult to define which privileges should cause the alternate prompt. However, a sufficiently powerful user should be reminded of that power by having an alternate prompt.

PS3 This variable is used by the KornShell for the *select* command. Since the POSIX shell does not include *select*, *PS3* was omitted.

PS4 This variable is used for shell debugging. For example, the following script:

```
PS4='[${LINENO}]+ '
set −x
echo Hello
```

writes the following to standard error:

```
[3]+ echo Hello
```

RANDOM This pseudo-random number generator was not seen as being useful to interactive users.

SECONDS Although this variable is sometimes used with *PS1* to allow the display of the current time in the prompt of the user, it is not one that would be manipulated frequently enough by an interactive user to include in the Shell and Utilities volume of IEEE Std. 1003.1-200x.

C.2.6 Word Expansions

Step (2) refers to the ‘‘portions of fields generated by step (1)’’. For example, if the word being expanded were `"$x+$y"` and *IFS*=+, the word would be split only if `"$x"` or `"$y"` contained `'+'`; the `'+'` in the original word was not generated by step (1).

IFS is used for performing field splitting on the results of parameter and command substitution; it is not used for splitting all fields. Previous versions of the shell used it for splitting all fields during field splitting, but this has severe problems because the shell can no longer parse its own script. There are also important security implications caused by this behavior. All useful applications of *IFS* use it for parsing input of the *read* utility and for splitting the results of parameter and command substitution.

The rule concerning expansion to a single field requires that if **foo**=**abc** and **bar**=**def**, that:

```
"$foo""$bar"
```

expands to the single field:

```
abcdef
```

The rule concerning empty fields can be illustrated by:

```
$    unset foo
$    set $foo bar '' xyz "$foo" abc
$    for i
>    do
>        echo "-$i-"
>    done
-bar-
--
-xyz-
--
-abc-
```

Step (1) indicates that parameter expansion, command substitution, and arithmetic expansion are all processed simultaneously as they are scanned. For example, the following is valid arithmetic:

```
x=1
echo $(( $(echo 3)+$x ))
```

An early proposal stated that tilde expansion preceded the other steps, but this is not the case in known historical implementations; if it were, and if a referenced home directory contained a `'$'` character, expansions would result within the directory name.

C.2.6.1 Tilde Expansion

Tilde expansion generally occurs only at the beginning of words, but an exception based on historical practice has been included:

```
PATH=/posix/bin:~dgk/bin
```

This is eligible for tilde expansion because tilde follows a colon and none of the relevant characters is quoted. Consideration was given to prohibiting this behavior because any of the following are reasonable substitutes:

```
PATH=$(printf %s ~karels/bin : ~bostic/bin)
```

```
for Dir in ~maart/bin ~srb/bin ...
do
    PATH=${PATH:+$PATH:}$Dir
done
```

In the first command, explicit colons are used for each directory. In all cases, the shell performs tilde expansion on each directory because all are separate words to the shell.

Note that expressions in operands such as:

```
make -k mumble LIBDIR=~chet/lib
```

do not qualify as shell variable assignments, and tilde expansion is not performed (unless the command does so itself, which *make* does not).

Because of the requirement that the word is not quoted, the following are not equivalent; only the last causes tilde expansion:

```
\~hlj/   ~h\lj/   ~"hlj"/   ~hlj\/   ~hlj/
```

In an early proposal, tilde expansion occurred following any unquoted equals sign or colon, but this was removed because of its complexity and to avoid breaking commands such as:

```
rcp hostname:~marc/.profile .
```

A suggestion was made that the special sequence `"$~"` should be allowed to force tilde expansion anywhere. Since this is not historical practice, it has been left for future implementations to evaluate. (The description in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.2, Quoting requires that a dollar sign be quoted to represent itself, so the `"$~"` combination is already unspecified.)

The results of giving tilde with an unknown login name are undefined because the KornShell `"~+"` and `"~-"` constructs make use of this condition, but in general it is an error to give an incorrect login name with tilde. The results of having *HOME* unset are unspecified because some historical shells treat this as an error.

C.2.6.2 Parameter Expansion

The rule for finding the closing `'}'` in `"${...}"` is the one used in the KornShell and is upwardly-compatible with the Bourne shell, which does not determine the closing `'}'` until the word is expanded. The advantage of this is that incomplete expansions, such as:

```
${foo
```

can be determined during tokenization, rather than during expansion.

The string length and substring capabilities were included because of the demonstrated need for them, based on their usage in other shells, such as C shell and KornShell.

Historical versions of the KornShell have not performed tilde expansion on the word part of parameter expansion; however, it is more consistent to do so.

C.2.6.3 Command Substitution

The `"$()"` form of command substitution solves a problem of inconsistent behavior when using backquotes. For example:

Command	Output
`echo ’\$x’`	`\$x`
`echo ‘echo ’\$x’‘`	`$x`
`echo $(echo ’\$x’)`	`\$x`

Additionally, the backquoted syntax has historical restrictions on the contents of the embedded command. While the newer "$()" form can process any kind of valid embedded script, the backquoted form cannot handle some valid scripts that include backquotes. For example, these otherwise valid embedded scripts do not work in the left column, but do work on the right:

```
echo ‘                             echo $(
cat <<\eof                         cat <<\eof
a here-doc with ‘                  a here-doc with )
eof                                eof
‘                                  )

echo ‘                             echo $(
echo abc # a comment with ‘        echo abc # a comment with )
‘                                  )

echo ‘                             echo $(
echo ’‘’                           echo ’)’
‘                                  )
```

Because of these inconsistent behaviors, the backquoted variety of command substitution is not recommended for new applications that nest command substitutions or attempt to embed complex scripts.

The KornShell feature:

> If *command* is of the form <*word*, *word* is expanded to generate a path name, and the value of the command substitution is the contents of this file with any trailing <newline>s deleted.

was omitted from the Shell and Utilities volume of IEEE Std. 1003.1-200x because $(*cat word*) is an appropriate substitute. However, to prevent breaking numerous scripts relying on this feature, it is unspecified to have a script within "$()" that has only redirections.

The requirement to separate "$(" and ’(’ when a single subshell is command-substituted is to avoid any ambiguities with arithmetic expansion.

C.2.6.4 Arithmetic Expansion

The "(())" form of KornShell arithmetic in early proposals was omitted. The standard developers concluded that there was a strong desire for some kind of arithmetic evaluator to replace *expr*, and that relating it to ’$’ makes it work well with the standard shell language, and it provides access to arithmetic evaluation in places where accessing a utility would be inconvenient.

The syntax and semantics for arithmetic were changed for the ISO/IEC 9945-2:1993 standard. The language is essentially a pure arithmetic evaluator of constants and operators (excluding assignment) and represents a simple subset of the previous arithmetic language (which was derived from the KornShell "(())" construct). The syntax was changed from that of a command denoted by ((*expression*)) to an expansion denoted by $((*expression*)). The new form is a dollar expansion (’$’) that evaluates the expression and substitutes the resulting value. Objections to the previous style of arithmetic included that it was too complicated, did not fit in well with the use of variables in the shell, and its syntax conflicted with subshells. The justification for the new syntax is that the shell is traditionally a macro language, and if a new feature is to be added, it should be accomplished by extending the capabilities presented by the current model of the shell, rather than by inventing a new one outside the model; adding a new dollar expansion was perceived to be the most intuitive and least destructive way to add such a new capability.

In early proposals, a form $[*expression*] was used. It was functionally equivalent to the `"$(())"` of the current text, but objections were lodged that the 1988 KornShell had already implemented `"$(())"` and there was no compelling reason to invent yet another syntax. Furthermore, the `"$[]"` syntax had a minor incompatibility involving the patterns in **case** statements.

The portion of the ISO C standard arithmetic operations selected corresponds to the operations historically supported in the KornShell.

It was concluded that the *test* command (**[**) was sufficient for the majority of relational arithmetic tests, and that tests involving complicated relational expressions within the shell are rare, yet could still be accommodated by testing the value of `"$(())"` itself. For example:

```
# a complicated relational expression
while [ $(( (($x + $y)/($a * $b)) < ($foo*$bar) )) -ne 0 ]
```

or better yet, the rare script that has many complex relational expressions could define a function like this:

```
val() {
    return $((!$1))
}
```

and complicated tests would be less intimidating:

```
while val $(( (($x + $y)/($a * $b)) < ($foo*$bar) ))
do
    # some calculations
done
```

A suggestion that was not adopted was to modify *true* and *false* to take an optional argument, and *true* would exit true only if the argument was non-zero, and *false* would exit false only if the argument was non-zero:

```
while true $(($x > 5 && $y <= 25))
```

There is a minor portability concern with the new syntax. The example $((2+2)) could have been intended to mean a command substitution of a utility named 2+2 in a subshell. The standard developers considered this to be obscure and isolated to some KornShell scripts (because `"$()"` command substitution existed previously only in the KornShell). The text on command substitution requires that the `"$("` and `'('` be separate tokens if this usage is needed.

An example such as:

```
echo $((echo hi);(echo there))
```

should not be misinterpreted by the shell as arithmetic because attempts to balance the parentheses pairs would indicate that they are subshells. However, as indicated by the Base Definitions volume of IEEE Std. 1003.1-200x, Section 3.115, Control Operator, a conforming application must separate two adjacent parentheses with white space to indicate nested subshells.

C.2.6.5 Field Splitting

The operation of field splitting using *IFS*, as described in early proposals, was based on the way the KornShell splits words, but it is incompatible with other common versions of the shell. However, each has merit, and so a decision was made to allow both. If the *IFS* variable is unset or is <space><tab><newline>, the operation is equivalent to the way the System V shell splits words. Using characters outside the <space><tab><newline> set yields the KornShell behavior, where each of the non-<space><tab><newline> characters is significant. This behavior, which affords the most flexibility, was taken from the way the original *awk* handled field splitting.

Rule (3) can be summarized as a pseudo-ERE:

```
(s*ns*|s+)
```

where *s* is an *IFS* white space character and *n* is a character in the *IFS* that is not white space. Any string matching that ERE delimits a field, except that the *s*+ form does not delimit fields at the beginning or the end of a line. For example, if *IFS* is <space>/<comma>/<tab>, the string:

```
<space><space>red<space><space>,<space>white<space>blue
```

yields the three colors as the delimited fields.

C.2.6.6 Path Name Expansion

There is no additional rationale for this section.

C.2.6.7 Quote Removal

There is no additional rationale for this section.

C.2.7 Redirection

In the System Interfaces volume of IEEE Std. 1003.1-200x, file descriptors are integers in the range 0–({OPEN_MAX}–1). The file descriptors discussed in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.7, Redirection are that same set of small integers.

Having multi-digit file descriptor numbers for I/O redirection can cause some obscure compatibility problems. Specifically, scripts that depend on an example command:

```
echo 22>/dev/null
```

echoing 2 to standard error or 22 to standard output are no longer portable. However, the file descriptor number still must be delimited from the preceding text. For example:

```
cat file2>foo
```

writes the contents of **file2**, not the contents of **file**.

The `">|"` format of output redirection was adopted from the KornShell. Along with the *noclobber* option, *set* –**C**, it provides a safety feature to prevent inadvertent overwriting of existing files. (See the RATIONALE for the *pathchk* utility for why this step was taken.) The restriction on regular files is historical practice.

The System V shell and the KornShell have differed historically on path name expansion of *word*; the former never performed it, the latter only when the result was a single field (file). As a compromise, it was decided that the KornShell functionality was useful, but only as a shorthand device for interactive users. No reasonable shell script would be written with a command such as:

```
cat foo > a*
```

Thus, shell scripts are prohibited from doing it, while interactive users can select the shell with which they are most comfortable.

The construct 2>&1 is often used to redirect standard error to the same file as standard output. Since the redirections take place beginning to end, the order of redirections is significant. For example:

```
ls > foo 2>&1
```

directs both standard output and standard error to file **foo**. However:

```
ls 2>&1 > foo
```

only directs standard output to file **foo** because standard error was duplicated as standard output before standard output was directed to file **foo**.

The "<>" operator could be useful in writing an application that worked with several terminals, and occasionally wanted to start up a shell. That shell would in turn be unable to run applications that run from an ordinary controlling terminal unless it could make use of "<>" redirection. The specific example is a historical version of the pager *more*, which reads from standard error to get its commands, so standard input and standard output are both available for their usual usage. There is no way of saying the following in the shell without "<>":

```
cat food | more − >/dev/tty03 2<>/dev/tty03
```

Another example of "<>" is one that opens **/dev/tty** on file descriptor 3 for reading and writing:

```
exec 3<> /dev/tty
```

An example of creating a lock file for a critical code region:

```
set −C
until    2> /dev/null > lockfile
do       sleep 30
done
set +C
perform critical function
rm lockfile
```

Since **/dev/null** is not a regular file, no error is generated by redirecting to it in *noclobber* mode.

Tilde expansion is not performed on a here-document because the data is treated as if it were enclosed in double quotes.

C.2.7.1 Redirecting Input

There is no additional rationale for this section.

C.2.7.2 Redirecting Output

There is no additional rationale for this section.

C.2.7.3 Appending Redirected Output

There is no additional rationale for this section.

C.2.7.4 Here-Document

There is no additional rationale for this section.

C.2.7.5 Duplicating an Input File Descriptor

There is no additional rationale for this section.

C.2.7.6 Duplicating an Output File Descriptor

There is no additional rationale for this section.

C.2.7.7 Open File Descriptors for Reading and Writing

There is no additional rationale for this section.

C.2.8 Exit Status and Errors

C.2.8.1 Consequences of Shell Errors

There is no additional rationale for this section.

C.2.8.2 Exit Status for Commands

There is a historical difference in *sh* and *ksh* non-interactive error behavior. When a command named in a script is not found, some implementations of *sh* exit immediately, but *ksh* continues with the next command. Thus, the Shell and Utilities volume of IEEE Std. 1003.1-200x says that the shell ''may'' exit in this case. This puts a small burden on the programmer, who has to test for successful completion following a command if it is important that the next command not be executed if the previous command was not found. If it is important for the command to have been found, it was probably also important for it to complete successfully. The test for successful completion would not need to change.

Historically, shells have returned an exit status of 128+*n*, where *n* represents the signal number. Since signal numbers are not standardized, there is no portable way to determine which signal caused the termination. Also, it is possible for a command to exit with a status in the same range of numbers that the shell would use to report that the command was terminated by a signal. Implementations are encouraged to choose exit values greater than 256 to indicate programs that terminate by a signal so that the exit status cannot be confused with an exit status generated by a normal termination.

Historical shells make the distinction between ''utility not found'' and ''utility found but cannot execute'' in their error messages. By specifying two seldomly used exit status values for these cases, 127 and 126 respectively, this gives an application the opportunity to make use of this distinction without having to parse an error message that would probably change from locale to locale. The *command*, *env*, *nohup*, and *xargs* utilities in the Shell and Utilities volume of IEEE Std. 1003.1-200x have also been specified to use this convention.

When a command fails during word expansion or redirection, most historical implementations exit with a status of 1. However, there was some sentiment that this value should probably be much higher so that an application could distinguish this case from the more normal exit status values. Thus, the language ''greater than zero'' was selected to allow either method to be implemented.

C.2.9 Shell Commands

A description of an ‘‘empty command’’ was removed from an early proposal because it is only relevant in the cases of *sh* **–c** "", *system*(""), or an empty shell-script file (such as the implementation of *true* on some historical systems). Since it is no longer mentioned in the Shell and Utilities volume of IEEE Std. 1003.1-200x, it falls into the silently unspecified category of behavior where implementations can continue to operate as they have historically, but conforming applications do not construct empty commands. (However, note that *sh* does explicitly state an exit status for an empty string or file.) In an interactive session or a script with other commands, extra <newline>s or semicolons, such as;

```
$ false
$
$ echo $?
1
```

would not qualify as the empty command described here because they would be consumed by other parts of the grammar.

C.2.9.1 Simple Commands

The enumerated list is used only when the command is actually going to be executed. For example, in:

```
true || $foo *
```

no expansions are performed.

The following example illustrates both how a variable assignment without a command name affects the current execution environment, and how an assignment with a command name only affects the execution environment of the command:

```
$ x=red
$ echo $x
red
$ export x
$ sh –c ’echo $x’
red
$ x=blue sh –c ’echo $x’
blue
$ echo $x
red
```

This next example illustrates that redirections without a command name are still performed:

```
$ ls foo
ls: foo: no such file or directory
$ > foo
$ ls foo
foo
```

A command without a command name, but one that includes a command substitution, has an exit status of the last command substitution that the shell performed. For example:

```
if      x=$(command)
then    ...
fi
```

An example of redirections without a command name being performed in a subshell shows that the here-document does not disrupt the standard input of the **while** loop:

```
IFS=:
while    read a b
do       echo $a
         <<-eof
         Hello
         eof
done </etc/passwd
```

Some examples of commands without command names in AND-OR lists:

```
> foo || {
    echo "error: foo cannot be created" >&2
    exit 1
}

# set saved if /vmunix.save exists
test -f /vmunix.save && saved=1
```

Command substitution and redirections without command names both occur in subshells, but they are not necessarily the same ones. For example, in:

```
exec 3> file
var=$(echo foo >&3) 3>&1
```

it is unspecified whether **foo** is echoed to the file or to standard output.

Command Search and Execution

This description requires that the shell can execute shell scripts directly, even if the underlying system does not support the common "#!" interpreter convention. That is, if file **foo** contains shell commands and is executable, the following executes **foo**:

```
./foo
```

The command search shown here does not match all historical implementations. A more typical sequence has been:

- Any built-in (special or regular)
- Functions
- Path search for executable files

But there are problems with this sequence. Since the programmer has no idea in advance which utilities might have been built into the shell, a function cannot be used to override portably a utility of the same name. (For example, a function named *cd* cannot be written for many historical systems.) Furthermore, the *PATH* variable is partially ineffective in this case, and only a path name with a slash can be used to ensure a specific executable file is invoked.

After the *execve*( ) failure described, the shell normally executes the file as a shell script. Some implementations, however, attempt to detect whether the file is actually a script and not an executable from some other architecture. The method used by the KornShell is allowed by the text that indicates non-text files may be bypassed.

The sequence selected for the Shell and Utilities volume of IEEE Std. 1003.1-200x acknowledges that special built-ins cannot be overridden, but gives the programmer full control over which versions of other utilities are executed. It provides a means of suppressing function lookup (via the *command* utility) for the user's own functions and ensures that any regular built-ins or functions provided by the implementation are under the control of the path search. The mechanisms for associating built-ins or functions with executable files in the path are not specified by the Shell and Utilities volume of IEEE Std. 1003.1-200x, but the wording requires that if either is implemented, the application is not able to distinguish a function or built-in from an executable (other than in terms of performance, presumably). The implementation ensures that all effects specified by the Shell and Utilities volume of IEEE Std. 1003.1-200x resulting from the invocation of the regular built-in or function (interaction with the environment, variables, traps, and so on) are identical to those resulting from the invocation of an executable file.

Examples

Consider three versions of the *ls* utility:

1. The application includes a shell function named *ls*.
2. The user writes a utility named *ls* and puts it in **/fred/bin**.
3. The example implementation provides *ls* as a regular shell built-in that is invoked (either by the shell or directly by *exec*) when the path search reaches the directory **/posix/bin**.

If *PATH*=**/posix/bin**, various invocations yield different versions of *ls*:

Invocation	**Version of *ls***
ls (from within application script)	(1) function
command ls (from within application script)	(3) built-in
ls (from within makefile called by application)	(3) built-in
system("*ls*")	(3) built-in
PATH="**/fred/bin**:$*PATH*" *ls*	(2) user's version

C.2.9.2 Pipelines

Because pipeline assignment of standard input or standard output or both takes place before redirection, it can be modified by redirection. For example:

```
$ command1 2>&1 | command2
```

sends both the standard output and standard error of *command1* to the standard input of *command2*.

The reserved word `!` allows more flexible testing using AND and OR lists.

It was suggested that it would be better to return a non-zero value if any command in the pipeline terminates with non-zero status (perhaps the bitwise-inclusive OR of all return values). However, the choice of the last-specified command semantics are historical practice and would cause applications to break if changed. An example of historical behavior:

```
$ sleep 5 | (exit 4)
$ echo $?
4
$ (exit 4) | sleep 5
$ echo $?
0
```

C.2.9.3 Lists

The equal precedence of `"&&"` and `"||"` is historical practice. The standard developers evaluated the model used more frequently in high-level programming languages, such as C, to allow the shell logical operators to be used for complex expressions in an unambiguous way, but they could not allow historical scripts to break in the subtle way unequal precedence might cause. Some arguments were posed concerning the `"{}"` or `"()"` groupings that are required historically. There are some disadvantages to these groupings:

- The `"()"` can be expensive, as they spawn other processes on some systems. This performance concern is primarily an implementation issue.

- The `"{}"` braces are not operators (they are reserved words) and require a trailing space after each `'{'`, and a semicolon before each `'}'`. Most programmers (and certainly interactive users) have avoided braces as grouping constructs because of the problematic syntax required. Braces were not changed to operators because that would generate compatibility issues even greater than the precedence question; braces appear outside the context of a keyword in many shell scripts.

Asynchronous Lists

The grammar treats a construct such as:

```
foo & bar & bam &
```

as one ‘‘asynchronous list’’, but since the status of each element is tracked by the shell, the term ‘‘element of an asynchronous list’’ was introduced to identify just one of the **foo**, **bar**, or **bam** portions of the overall list.

Unless the implementation has an internal limit, such as {CHILD_MAX}, on the retained process IDs, it would require unbounded memory for the following example:

```
while true
do      foo & echo $!
done
```

The treatment of the signals SIGINT and SIGQUIT with asynchronous lists is described in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.12, Signals and Error Handling.

Since the connection of the input to the equivalent of **/dev/null** is considered to occur before redirections, the following script would produce no output:

```
exec < /etc/passwd
cat <&0 &
wait
```

Sequential Lists

There is no additional rationale for this section.

AND Lists

There is no additional rationale for this section.

OR Lists

There is no additional rationale for this section.

C.2.9.4 Compound Commands

Grouping Commands

The semicolon shown {*compound-list*;} is an example of a control operator delimiting the **}** reserved word. Other delimiters are possible, as shown in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.11, Shell Grammar; <newline> is frequently used.

A proposal was made to use the **<do-done>** construct in all cases where command grouping in the current process environment is performed, identifying it as a construct for the grouping commands, as well as for shell functions. This was not included because the shell already has a grouping construct for this purpose (`"{}"`), and changing it would have been counter-productive.

For Loop

The format is shown with generous usage of <newline>s. See the grammar in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.11, Shell Grammar for a precise description of where <newline>s and semicolons can be interchanged.

Some historical implementations support `'{'` and `'}'` as substitutes for **do** and **done**. The standard developers chose to omit them, even as an obsolescent feature. (Note that these substitutes were only for the **for** command; the **while** and **until** commands could not use them historically because they are followed by compound-lists that may contain `"{...}"` grouping commands themselves.)

The reserved word pair **do** ... **done** was selected rather than **do** ... **od** (which would have matched the spirit of **if** ... **fi** and **case** ... **esac**) because *od* is already the name of a standard utility.

PASC Interpretation 1003.2 #169 has been applied changing the grammar.

Case Conditional Construct

An optional left parenthesis before *pattern* was added to allow numerous historical KornShell scripts to conform. At one time, using the leading parenthesis was required if the **case** statement was to be embedded within a `"$()"` command substitution; this is no longer the case with the POSIX shell. Nevertheless, many historical scripts use the left parenthesis, if only because it makes matching-parenthesis searching easier in *vi* and other editors. This is a relatively simple implementation change that is upward-compatible for all scripts.

Consideration was given to requiring *break* inside the *compound-list* to prevent falling through to the next pattern action list. This was rejected as being nonexisting practice. An interesting undocumented feature of the KornShell is that using `";&"` instead of `";;"` as a terminator causes the exact opposite behavior—the flow of control continues with the next *compound-list*.

The pattern `'*'`, given as the last pattern in a **case** construct, is equivalent to the default case in a C-language **switch** statement.

The grammar shows that reserved words can be used as patterns, even if one is the first word on a line. Obviously, the reserved word **esac** cannot be used in this manner.

If Conditional Construct

The precise format for the command syntax is described in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.11, Shell Grammar.

While Loop

The precise format for the command syntax is described in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.11, Shell Grammar.

Until Loop

The precise format for the command syntax is described in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.11, Shell Grammar.

C.2.9.5 Function Definition Command

The description of functions in an early proposal was based on the notion that functions should behave like miniature shell scripts; that is, except for sharing variables, most elements of an execution environment should behave as if they were a new execution environment, and changes to these should be local to the function. For example, traps and options should be reset on entry to the function, and any changes to them do not affect the traps or options of the caller. There were numerous objections to this basic idea, and the opponents asserted that functions were intended to be a convenient mechanism for grouping common commands that were to be executed in the current execution environment, similar to the execution of the *dot* special built-in.

It was also pointed out that the functions described in that early proposal did not provide a local scope for everything a new shell script would, such as the current working directory, or *umask*, but instead provided a local scope for only a few select properties. The basic argument was that if a local scope is needed for the execution environment, the mechanism already existed: the application can put the commands in a new shell script and call that script. All historical shells that implemented functions, other than the KornShell, have implemented functions that operate in the current execution environment. Because of this, traps and options have a global scope within a shell script. Local variables within a function were considered and included in another early proposal (controlled by the special built-in *local*), but were removed because they do not fit the simple model developed for functions and because there was some opposition to adding yet another new special built-in that was not part of historical practice. Implementations should reserve the identifier *local* (as well as *typeset*, as used in the KornShell) in case this local variable mechanism is adopted in a future version of IEEE Std. 1003.1-200x.

A separate issue from the execution environment of a function is the availability of that function to child shells. A few objectors maintained that just as a variable can be shared with child shells by exporting it, so should a function. In early proposals, the *export* command therefore had a –**f** flag for exporting functions. Functions that were exported were to be put into the environment as *name*( )=*value* pairs, and upon invocation, the shell would scan the environment for these and automatically define these functions. This facility was strongly opposed and was omitted. Some of the arguments against exportable functions were as follows:

- There was little historical practice. The Ninth Edition shell provided them, but there was controversy over how well it worked.

- There are numerous security problems associated with functions appearing in the environment of a user and overriding standard utilities or the utilities owned by the application.
- There was controversy over requiring *make* to import functions, where it has historically used an *exec* function for many of its command line executions.
- Functions can be big and the environment is of a limited size. (The counter-argument was that functions are no different than variables in terms of size: there can be big ones, and there can be small ones—and just as one does not export huge variables, one does not export huge functions. However, this might not apply to the average shell-function writer, who typically writes much larger functions than variables.)

As far as can be determined, the functions in the Shell and Utilities volume of IEEE Std. 1003.1-200x match those in System V. Earlier versions of the KornShell had two methods of defining functions:

```
function fname { compound-list }
```

and:

```
fname() { compound-list }
```

The latter used the same definition as the Shell and Utilities volume of IEEE Std. 1003.1-200x, but differed in semantics, as described previously. The current edition of the KornShell aligns the latter syntax with the Shell and Utilities volume of IEEE Std. 1003.1-200x and keeps the former as is.

The name space for functions is limited to that of a *name* because of historical practice. Complications in defining the syntactic rules for the function definition command and in dealing with known extensions such as the `"@()"` usage in the KornShell prevented the name space from being widened to a *word*. Using functions to support synonyms such as the `"!!"` and `'%'` usage in the C shell is thus disallowed to portable applications, but acceptable as an extension. For interactive users, the aliasing facilities in the Shell and Utilities volume of IEEE Std. 1003.1-200x should be adequate for this purpose. It is recognized that the name space for utilities in the file system is wider than that currently supported for functions, if the portable file name character set guidelines are ignored, but it did not seem useful to mandate extensions in systems for so little benefit to portable applications.

The `"()"` in the function definition command consists of two operators. Therefore, intermixing <blank>s with the *fname*, `'('`, and `')'` is allowed, but unnecessary.

An example of how a function definition can be used wherever a simple command is allowed:

```
# If variable i is equal to "yes",
# define function foo to be ls −l
#
[ "$i" = yes ] && foo() {
    ls −l
}
```

C.2.10 Executable Script

The working group did not reach consensus to adopt this as a core requirement—that is, for POSIX-conforming applications—however, existing practice on UNIX systems indicated that it should be added as an XSI extension, and this was brought into the scope of this revision by The Open Group Base Resolution bwg2000-004. The scope of this feature is to document existing practice and not to invent.

Applications must not assume that the standard utilities will be available in any particular named directory. For example, it cannot be assumed that standard versions of *awk* and *sh* will be available as **/bin/sh** or **/bin/awk**, respectively, since implementations are permitted to provide non-standard versions of the utilities in these directories.

It is recommended that an installation script for executable scripts use the standard *PATH* returned by a call to the *getconf* utility with the argument *PATH*, combined with the *command* utility to determine the location of a standard utility.

For example, to determine the location of the standard *sh* utility:

```
command -v sh
```

On some systems this might return:

```
/usr/xpg4/bin/sh
```

Note that the installation script should ensure that the returned path name is an absolute path name prior to use, since a shell built-in might be returned for some utilities.

C.2.11 Shell Grammar

There are several subtle aspects of this grammar where conventional usage implies rules about the grammar that in fact are not true.

For *compound_list*, only the forms that end in a *separator* allow a reserved word to be recognized, so usually only a *separator* can be used where a compound list precedes a reserved word (such as **Then**, **Else**, **Do** and **Rbrace**). Explicitly requiring a separator would disallow such valid (if rare) statements as:

```
if (false) then (echo x) else (echo y) fi
```

See the Note under special grammar rule 1.

Concerning the third sentence of rule (1) (''Also, if the parser ...''):

- This sentence applies rather narrowly: when a compound list is terminated by some clear delimiter (such as the closing **fi** of an inner **if_clause**) then it would apply; where the compound list might continue (as in after a `';'`), rule (7a) (and consequently the first sentence of rule (1)) would apply. In many instances the two conditions are identical, but this part of rule (1) does not give license to treating a **WORD** as a reserved word unless it is in a place where a reserved word has to appear.
- The statement is equivalent to requiring that when the LR(1) lookahead set contains exactly one reserved word, it must be recognized if it is present. (Here ''LR(1)'' refers to the theoretical concepts, not to any real parser generator.)

  For example, in the construct below, and when the parser is at the point marked with `'^'`, the only next legal token is **then** (this follows directly from the grammar rules):

  ```
  if if...fi then ... fi
             ^
  ```

At that point, the **then** must be recognized as a reserved word.

(Depending on the parser generator actually used, ‘‘extra’’ reserved words may be in some lookahead sets. It does not really matter if they are recognized, or even if any possible reserved word is recognized in that state, because if it is recognized and is not in the (theoretical) LR(1) lookahead set, an error is ultimately detected. In the example above, if some other reserved word (for example, **while**) is also recognized, an error occurs later.

This is approximately equivalent to saying that reserved words are recognized after other reserved words (because it is after a reserved word that this condition occurs), but avoids the ‘‘except for ...’’ list that would be required for **case**, **for**, and so on. (Reserved words are of course recognized anywhere a *simple_command* can appear, as well. Other rules take care of the special cases of non-recognition, such as rule (4) for **case** statements.)

Note that the body of here-documents are handled by token recognition (see the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.3, Token Recognition) and do not appear in the grammar directly. (However, the here-document I/O redirection operator is handled as part of the grammar.)

The start symbol of the grammar (**complete_command**) represents either input from the command line or a shell script. It is repeatedly applied by the interpreter to its input and represents a single ‘‘chunk’’ of that input as seen by the interpreter.

C.2.11.1 Shell Grammar Lexical Conventions

There is no additional rationale for this section.

C.2.11.2 Shell Grammar Rules

There is no additional rationale for this section.

C.2.12 Signals and Error Handling

There is no additional rationale for this section.

C.2.13 Shell Execution Environment

Some systems have implemented the last stage of a pipeline in the current environment so that commands such as:

```
command | read foo
```

set variable **foo** in the current environment. This extension is allowed, but not required; therefore, a shell programmer should consider a pipeline to be in a subshell environment, but not depend on it.

In early proposals, the description of execution environment failed to mention that each command in a multiple command pipeline could be in a subshell execution environment. For compatibility with some historical shells, the wording was phrased to allow an implementation to place any or all commands of a pipeline in the current environment. However, this means that a POSIX application must assume each command is in a subshell environment, but not depend on it.

The wording about shell scripts is meant to convey the fact that describing ‘‘trap actions’’ can only be understood in the context of the shell command language. Outside of this context, such as in a C-language program, signals are the operative condition, not traps.

C.2.14 Pattern Matching Notation

Pattern matching is a simpler concept and has a simpler syntax than REs, as the former is generally used for the manipulation of file names, which are relatively simple collections of characters, while the latter is generally used to manipulate arbitrary text strings of potentially greater complexity. However, some of the basic concepts are the same, so this section points liberally to the detailed descriptions in the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 9, Regular Expressions.

C.2.14.1 Patterns Matching a Single Character

Both quoting and escaping are described here because pattern matching must work in three separate circumstances:

1. Calling directly upon the shell, such as in path name expansion or in a **case** statement. All of the following match the string or file **abc**:

   ```
   abc "abc" a"b"c a\bc a[b]c a["b"]c a[\b]c a["\b"]c a?c a*c
   ```

   The following do not:

   ```
   "a?c" a\*c a\[b]c
   ```

2. Calling a utility or function without going through a shell, as described for *find* and the *fnmatch*( ) function defined in the System Interfaces volume of IEEE Std. 1003.1-200x.

3. Calling utilities such as *find*, *cpio*, *tar*, or *pax* through the shell command line. In this case, shell quote removal is performed before the utility sees the argument. For example, in:

   ```
   find /bin -name "e\c[\h]o" -print
   ```

   after quote removal, the backslashes are presented to *find* and it treats them as escape characters. Both precede ordinary characters, so the *c* and *h* represent themselves and *echo* would be found on many historical systems (that have it in **/bin**). To find a file name that contained shell special characters or pattern characters, both quoting and escaping are required, such as:

   ```
   pax -r ... "*a\(\?"
   ```

   to extract a file name ending with `"a(?"`.

Conforming applications are required to quote or escape the shell special characters (sometimes called metacharacters). If used without this protection, syntax errors can result or implementation extensions can be triggered. For example, the KornShell supports a series of extensions based on parentheses in patterns.

The restriction on a circumflex in a bracket expression is to allow implementations that support pattern matching using the circumflex as the negation character in addition to the exclamation mark. A portable application must use something like `"[\^!]"` to match either character.

C.2.14.2 Patterns Matching Multiple Characters

Since each asterisk matches zero or more occurrences, the patterns `"a*b"` and `"a**b"` have identical functionality.

Examples

`a[bc]`	Matches the strings "ab" and "ac".
`a*d`	Matches the strings "ad", "abd", and "abcd", but not the string "abc".
`a*d*`	Matches the strings "ad", "abcd", "abcdef", "aaaad", and "adddd".
`*a*d`	Matches the strings "ad", "abcd", "efabcd", "aaaad", and "adddd".

C.2.14.3 Patterns Used for File Name Expansion

The caveat about a slash within a bracket expression is derived from historical practice. The pattern "a[b/c]d" does not match such path names as **abd** or **a/d**. On some systems (including those conforming to the Single UNIX Specification), it matched a path name of literally "a[b/c]d". On other systems, it produced an undefined condition (an unescaped '[' used outside a bracket expression). In this version, the XSI behavior is now required.

File names beginning with a period historically have been specially protected from view on UNIX systems. A proposal to allow an explicit period in a bracket expression to match a leading period was considered; it is allowed as an implementation extension, but a conforming application cannot make use of it. If this extension becomes popular in the future, it will be considered for a future version of the Shell and Utilities volume of IEEE Std. 1003.1-200x.

Historical systems have varied in their permissions requirements. To match **f*/bar** has required read permissions on the **f*** directories in the System V shell, but the Shell and Utilities volume of IEEE Std. 1003.1-200x, the C shell, and KornShell require only search permissions.

C.2.15 Special Built-In Utilities

See the RATIONALE sections on the individual reference pages.

C.3 Batch Environment Services and Utilities

Scope of the Batch Environment Option

This section summarizes the deliberations of the IEEE P1003.15 (Batch Environment) working group in the development of the Batch Environment option, which covers a set of services and utilities defining a batch processing system.

This informative section contains historical information concerning the contents of the amendment and describes why features were included or discarded by the working group.

History of Batch Systems

The supercomputing technical committee began as a ‘‘Birds Of a Feather’’ (BOF) at the January 1987 Usenix meeting. There was enough general interest to form a supercomputing attachment to the /usr/group working groups. Several subgroups rapidly formed. Of those subgroups, the batch group was the most ambitious. The first early meetings were spent evaluating user needs and existing batch implementations.

To evaluate user needs, individuals from the supercomputing community came and presented their needs. Common requests were flexibility, interoperability, control of resources, and ease-of-use. Backwards-compatibility was not an issue. The working group then evaluated some existing systems. The following different systems were evaluated:

- PROD
- Convex Distributed Batch
- NQS
- CTSS
- MDQS from Ballistics Research Laboratory (BRL)

Finally, NQS was chosen as a model because it satisfied not only the most user requirements, but because it was public domain, already implemented on a variety of hardware platforms, and networked-based.

Historical Implementations of Batch Systems

Deferred processing of work under the control of a scheduler has been a feature of most proprietary operating systems from the earliest days of multi-user systems in order to maximize utilization of the computer.

The arrival of UNIX systems proved to be a dilemma to many hardware providers and users because it did not include the sophisticated batch facilities offered by the proprietary systems. This omission was rectified in 1986 by NASA Ames Research Center who developed the Network Queuing System (NQS) as a portable UNIX application that allowed the routing and processing of batch ‘‘jobs’’ in a network. To encourage its usage, the product was later put into the public domain. It was promptly picked up by UNIX hardware providers, and ported and developed for their respective hardware and UNIX implementations.

Many major vendors, who traditionally offer a batch-dominated environment, ported the public-domain product to their systems, customized it to support the capabilities of their systems, and added many customer-requested features.

Due to the strong hardware provider and customer acceptance of NQS, it was decided to use NQS as the basis for the POSIX Batch Environment amendment in 1987. Other batch systems considered at the time included CTSS, MDQS (a forerunner of NQS from the Ballistics Research

Laboratory), and PROD (a Los Alamos Labs development). None were thought to have both the functionality and acceptability of NQS.

NQS Differences from the at utility

The base standard *at* and *batch* utilities are not sufficient to meet the batch processing needs in a supercomputing environment and additional functionality in the areas of resource management, job scheduling, system management, and control of output is required.

Batch Environment Option Definitions

The concept of a batch job is closely related to a session with a session leader. The main difference is that a batch job does not have a controlling terminal. There has been much debate over whether to use the term *request* or *job*. Job was the final choice because of the historical use of this term in the batch environment.

The current definition for job identifiers is not sufficient with the model of destinations. The current definition is:

```
sequence_number.originating_host
```

Using the model of destination, a host may include multiple batch nodes, the location of which is identified uniquely by a name or directory service. If the current definition is used, batch nodes running on the same host would have to coordinate their use of sequence numbers, as sequence numbers are assigned by the originating host. The alternative is to use the originating batch node name instead of the originating host name.

The reasons for wishing to run more than one batch system per host could be the following:

A test and production batch system are maintained on a single host. This is most likely in a development facility, but could also arise when a site is moving from one version to another. The new batch system could be installed as a test version that is completely separate from the production batch system, so that problems can be isolated to the test system. Requiring the batch nodes to coordinate their use of sequence numbers creates a dependency between the two nodes, and that defeats the purpose of running two nodes.

A site has multiple departments using a single host, with different management policies. An example of contention might be in job selection algorithms. One group might want a FIFO type of selection, while another group wishes to use a more complex algorithm based on resource availability. Again, requiring the batch nodes to coordinate is an unnecessary binding.

The proposal eventually accepted was to replace originating host with originating batch node. This supplies sufficient granularity to ensure unique job identifiers. If more than one batch node is on a particular host, they each have their own unique name.

The queue portion of a destination is not part of the job identifier as these are not required to be unique between batch nodes. For instance, two batch nodes may both have queues called small, medium, and large. It is only the batch node name that is uniquely identifiable throughout the batch system. The queue name has no additional function in this context.

Assume there are three batch nodes, each of which has its own name server. On batch node one, there are no queues. On batch node two, there are fifty queues. On batch node three, there are forty queues. The system administrator for batch node one does not have to configure queues, because there are none implemented. However, if a user wishes to send a job to either batch node two or three, the system administrator for batch node one must configure a destination that maps to the appropriate batch node and queue. If every queue is to be made accessible from batch node one, the system administrator has to configure ninety destinations.

To avoid requiring this, there should be a mechanism to allow a user to separate the destination into a batch node name and a queue name. Then, an implementation that is configured to get to all the batch nodes does not need any more configuration to allow a user to get to all of the queues on all of the batch nodes. The node name is used to locate the batch node, while the queue name is sent unchanged to that batch node.

The following are requirements that a destination identifier must be capable of providing:

- The ability to direct a job to a queue in a particular batch node.
- The ability to direct a job to a particular batch node.
- The ability to group at a higher level than just one queue. This includes grouping similar queues across multiple batch nodes (this is a pipe queue today).
- The ability to group batch nodes. This allows a user to submit a job to a group name with no knowledge of the batch node configuration. This also provides aliasing as a special case. Aliasing is a group containing only one batch node name. The group name is the alias.

In addition, the administrator has the following requirements:

- The ability to control access to the queues.
- The ability to control access to the batch nodes.
- The ability to control access to groups of queues (pipe queues).
- The ability to configure retry time intervals and durations.

The requirements of the user are met by destination as explained in the following:

The user has the ability to specify a queue name, which is known only to the batch node specified. There is no configuration of these queues required on the submitting node.

The user has the ability to specify a batch node whose name is network-unique. The configuration required is that the batch node be defined as an application, just as other applications such as FTP are configured.

Once a job reaches a queue, it can again become a user of the batch system. The batch node can choose to send the job to another batch node or queue or both. In other words, the routing is at an application level, and it is up to the batch system to choose where the job will be sent. Configuration is up to the batch node where the queue resides. This provides grouping of queues across batch nodes or within a batch node. The user submits the job to a queue, which by definition routes the job to other queues or nodes or both.

A node name may be given to a naming service, which returns multiple addresses as opposed to just one. This provides grouping at a batch node level. This is a local issue, meaning that the batch node must choose only one of these addresses. The list of addresses is not sent with the job, and once the job is accepted on another node, there is no connection between the list and the job. The requirements of the administrator are met by destination as explained in the following:

The control of queues is a batch system issue, and will be done using the batch administrative utilities.

The control of nodes is a network issue, and will be done through whatever network facilities are available.

The control of access to groups of queues (pipe queues) is covered by the control of any other queue. The fact that the job may then be sent to another destination is not relevant.

The propagation of a job across more than one point-to-point connection was dropped because of its complexity and because all of the issues arising from this capability could not be resolved.

It could be provided as additional functionality at some time in the future.

The addition of *network* as a defined term was done to clarify the difference between a network of batch nodes as opposed to a network of hosts. A network of batch nodes is referred to as a batch system. The network refers to the actual host configuration. A single host may have multiple batch nodes.

In the absence of a standard network naming convention, this option establishes its own convention for the sake of consistency and expediency. This is subject to change, should a future working group develop a standard naming convention for network path names.

C.3.1 Batch General Concepts

During the development of the Batch Environment option, a number of topics were discussed at length which influenced the wording of the normative text but could not be included in the final text. The following items are some of the most significant terms and concepts of those discussed:

- Small and Consistent Command Set

  Often, conventional utilities from UNIX systems have a very complicated utility syntax and usage. This can often result in confusion and errors when trying to use them. The Batch Environment option utility set, on the other hand, has been paired to a small set of robust utilities with an orthogonal calling sequence.

- Checkpoint/Restart

  This feature permits an already executing process to checkpoint or save its contents. Some implementations permit this at both the batch utility level; for example, checkpointing this job upon its abnormal termination or from within the job itself via a system call. Support of checkpoint/restart is optional. A conscious, careful effort was made to make the *qsub* and *qmgr* utilities consistently refer to checkpoint/restart as optional functionality.

- Rerunability

  When a user submits a job for batch processing, they can designate it ''rerunnable'' in that it will automatically resume execution from the start of the job if the machine on which it was executing crashes for some reason. The decision on whether the job will be rerun or not is entirely up to the submitter of the job and no decisions will be made within the batch system. A job that is rerunnable and has been submitted with the proper checkpoint/restart switch will first be checkpointed and execution begun from that point. Furthermore, use of the implementation-defined checkpoint/restart feature will be not be defined in this context.

- Error Codes

  All utilities exit with error status zero (0) if successful, one (1) if a user error occurred, and two (2) for an internal Batch Environment option error.

- Level of Portability

  Portability is specified at both the user, operator, and administrator levels. A conforming batch implementation prevents identical functionality and behavior at all these levels. Additionally, portable batch shell scripts with embedded Batch Environment option utilities adds an additional level of portability.

- Resource Specification

  A small set of globally understood resources, such as memory and CPU time, is specified. All conforming batch implementations are able to process them in a manner consistent with the yet-to-be-developed resource management model. Resources not in this amendment set are ignored and passed along as part of the argument stream of the utility.

- Maximum of 80 Characters on Output Display

  At one time, existing displays were limited to 80 characters in length for purposes of readability, both in the amendment and online. Current internationalization efforts discourage specifying displays in the normative text. Instead, all suggested displays appear in an informative annex for illustrative purposes. As before, length is limited to 80 characters for readability purposes.

- Queue Position

  Queue position is the place a job occupies in a queue. It is dependent on a variety of factors such as submission time and priority. Since priority may be affected by the implementation of fair share scheduling, the definition of queue position is implementation-defined.

- Queue ID

  A numerical queue ID is an external requirement for purposes of accounting. The identification number was chosen over queue name for processing convenience.

- Job ID

  A common notion of ''jobs'' is a collection of processes whose process group cannot be altered and is used for resource management and accounting. This concept is implementation-defined and, as such, has been omitted from the batch amendment.

- Bytes *versus* Words

  Except for one case, bytes are used as the standard unit for memory size. Furthermore, the definition of a word varies from machine to machine. Therefore, bytes will be the default unit of memory size.

- Regular Expressions

  The standard definition of regular expressions is much too broad to be used in the batch utility syntax. All that is needed is a simple concept of ''all''; for example, delete all my jobs from the named queue. For this reason, regular expressions have been eliminated from the batch amendment.

- Display Privacy

  How much data should be displayed locally through library functions? Local policy dictates the amount of privacy. Library functions must be used to create and enforce local policy. Network and local *qstat*s must reflect the policy of the server machine.

- Remote Host Naming Convention

  It was decided that host names would be a maximum of 255 characters in length, with at most 15 characters being shown in displays. The 255 character limit was chosen because it is consistent with BSD). The 15-character limit was an arbitrary decision.

- Network Administration

  Network administration is important, but is outside the scope of the batch amendment. Network administration could done with *rsh*. However, authentication becomes two-sided.

- Network Administration Philosophy

  Keep it simple. Centralized management should be possible. For example, Los Alamos needs a dumb set of CPUs to be managed by a central system *versus* several independently-managed systems as is the general case for the Batch Environment option.

- Operator Utility Defaults (that is, Default Host, User, Account, and so on)

  It was decided that usability would override orthogonality and syntactic consistency.

- The Batch System Manager and Operator Distinction

  The distinction between manager and operator is that operators can only control the flow of jobs. A manager can alter the batch system configuration in addition to job flow. POSIX makes a distinction between user and system administrator but goes no further. The concepts of manager and operator privileges fall under local policy. The distinction between manager and operator is historical in batch environments, and the Batch Environment option has continued that distinction.

- The Batch System Administrator

  An administrator is equivalent to a batch system manager.

- Network Administration *versus* Checkpoint/Recovery

  Network administration is better left for a future revision of IEEE Std. 1003.1-200x. If network administration is put up against checkpoint/recovery, the argument is that there are possible solutions for network administration. However, checkpoint/recovery currently has no solution. This is another reason the issue of checkpoint/recovery should be addressed first.

C.3.2 Batch Services

This rationale is provided as informative rather than normative text, to avoid placing requirements on implementors regarding the use of symbolic constants, but at the same time to give implementors a preferred practice for assigning values to these constants to promote interoperability.

The *Checkpoint* and *Minimum_Cpu_Interval* attributes induce a variety of behavior depending upon their values. Some jobs cannot or should not be checkpointed. Other users will simply need to ensure job continuation across planned downtimes; for example, scheduled preventive maintenance. For users consuming expensive resources, or for jobs that run longer than the mean time between failures, however, periodic checkpointing may be essential. However, system administrators must be able to set minimum checkpoint intervals on a queue-by-queue basis to guard against; for example, naive users specifying interval values too small on memory intensive jobs. Otherwise, system overhead would adversely affect performance.

The use of symbolic constants, such as NO_CHECKPOINT, was introduced to lend a degree of formalism and portability to this option.

Support for checkpointing is optional for servers. However, clients must provide for the −**c** option, since in a distributed environment the job may run on a server that does provide such support, even if the host of the client does not support the checkpoint feature.

If the user does not specify the −**c** option, the default action is left unspecified by this option. Some implementations may wish to do checkpointing by default; others may wish to checkpoint only under an explicit request from the user.

The *Priority* attribute has been made non-optional. All clients already had been required to support the −**p** option. The concept of prioritization is common in historical implementations. The default priority is left to the server to establish.

The *Hold_Types* attribute has been modified to allow for implementation-defined hold types to be passed to a batch server.

It was the intent of the IEEE P1003.15 working group to mandate the support for the *Resource_List* attribute in this option by referring to another amendment, specifically P1003.1a. However, during the development of P1003.1a this was excluded. As such this requirement has been removed from the normative text.

The *Shell_Path* attribute has been modified to accept a list of shell paths that are associated with a host. The name of the attribute has been changed to *Shell_Path_List*.

C.3.3 Common Behavior for Batch Environment Utilities

This section was defined to meet the goal of a ''Small and Consistent Command Set'' for this option.

C.4 Utilities

See the RATIONALE sections on the individual reference pages.

Rationale (Informative)

Part D:

Portability Considerations

The Open Group

Appendix D

Portability Considerations (Informative)

This section contains information to satisfy various international requirements:

- Section D.1 describes perceived user requirements.
- Section D.2 (on page 3560) indicates how the facilities of IEEE Std. 1003.1-200x satisfy those requirements.
- Section D.3 (on page 3567) offers guidance to writers of profiles on how the configurable options, limits, and optional behavior of IEEE Std. 1003.1-200x should be cited in profiles.

D.1 User Requirements

This section describes the user requirements that were perceived by the developers of IEEE Std. 1003.1-200x. The primary source for these requirements was an analysis of historical practice in widespread use, as typified by the base documents listed in Section A.1.1 (on page 3311).

IEEE Std. 1003.1-200x addresses the needs of users requiring open systems solutions for source code portability of applications. It currently addresses users requiring open systems solutions for source-code portability of applications involving multi-programming and process management (creating processes, signaling, and so on); access to files and directories in a hierarchy of file systems (opening, reading, writing, deleting files, and so on); access to asynchronous communications ports and other special devices; access to information about other users of the system; facilities supporting applications requiring bounded (realtime) response.

The following users are identified for IEEE Std. 1003.1-200x:

- Those employing applications written in high-level languages, such as C, Ada, or FORTRAN.
- Users who desire portable applications that do not necessarily require the characteristics of high-level languages (for example, the speed of execution of compiled languages or the relative security of source code intellectual property inherent in the compilation process).
- Users who desire portable applications that can be developed quickly and can be modified readily without the use of compilers and other system components that may be unavailable on small systems or those without special application development capabilities.
- Users who interact with a system to achieve general-purpose time-sharing capabilities common to most business or government offices or academic environments: editing, filing, inter-user communications, printing, and so on.
- Users who develop applications for POSIX-conformant systems.
- Users who develop applications for UNIX systems.

An acknowledged restriction on applicable users is that they are limited to the group of individuals who are familiar with the style of interaction characteristic of historically-derived systems based on one of the UNIX operating systems (as opposed to other historical systems with different models, such as MS/DOS, Macintosh, VMS, MVS, and so on). Typical users would include program developers, engineers, or general-purpose time-sharing users.

The requirements of users of IEEE Std. 1003.1-200x can be summarized as a single goal: *application source portability*. The requirements of the user are stated in terms of the requirements of portability of applications. This in turn becomes a requirement for a standardized set of syntax and semantics for operations commonly found on many operating systems.

The following sections list the perceived requirements for application portability.

D.1.1 Configuration Interrogation

An application must be able to determine whether and how certain optional features are provided and to identify the system upon which it is running, so that it may appropriately adapt to its environment.

Applications must have sufficient information to adapt to varying behaviors of the system.

D.1.2 Process Management

An application must be able to manage itself, either as a single process or as multiple processes. Applications must be able to manage other processes when appropriate.

Applications must be able to identify, control, create, and delete processes, and there must be communication of information between processes and to and from the system.

Applications must be able to use multiple flows of control with a process (threads) and synchronize operations between these flows of control.

D.1.3 Access to Data

Applications must be able to operate on the data stored on the system, access it, and transmit it to other applications. Information must have protection from unauthorized or accidental access or modification.

D.1.4 Access to the Environment

Applications must be able to access the external environment to communicate their input and results.

D.1.5 Access to Determinism and Performance Enhancements

Applications must have sufficient control of resource allocation to ensure the timeliness of interactions with external objects.

D.1.6 Operating System-Dependent Profile

The capabilities of the operating system may make certain optional characteristics of the base language in effect no longer optional, and this should be specified.

D.1.7 I/O Interaction

The interaction between the C language I/O subsystem (*stdio*) and the I/O subsystem of IEEE Std. 1003.1-200x must be specified.

D.1.8 Internationalization Interaction

The effects of the environment of IEEE Std. 1003.1-200x on the internationalization facilities of the C language must be specified.

D.1.9 C-Language Extensions

Certain functions in the C language must be extended to support the additional capabilities provided by IEEE Std. 1003.1-200x.

D.1.10 Command Language

Users should be able to define procedures that combine simple tools and/or applications into higher-level components that perform to the specific needs of the user. The user should be able to store, recall, use, and modify these procedures. These procedures should employ a powerful command language that is used for recurring tasks in portable applications (scripts) in the same way that it is used interactively to accomplish one-time tasks. The language and the utilities that it uses must be consistent between systems to reduce errors and retraining.

D.1.11 Interactive Facilities

Use the system to accomplish individual tasks at an interactive terminal. The interface should be consistent, intuitive, and offer usability enhancements to increase the productivity of terminal users, reduce errors, and minimize retraining costs. Online documentation or usage assistance should be available.

D.1.12 Accomplish Multiple Tasks Simultaneously

Access applications and interactive facilities from a single terminal without requiring serial execution: switch between multiple interactive tasks; schedule one-time or periodic background work; display the status of all work in progress or scheduled; influence the priority scheduling of work, when authorized.

D.1.13 Complex Data Manipulation

Manipulate data in files in complex ways: sort, merge, compare, translate, edit, format, pattern match, select subsets (strings, columns, fields, rows, and so on). These facilities should be available to both portable applications and interactive users.

D.1.14 File Hierarchy Manipulation

Create, delete, move/rename, copy, backup/archive, and display files and directories. These facilities should be available to both portable applications and interactive users.

D.1.15 Locale Configuration

Customize applications and interactive sessions for the cultural and language conventions of the user. Employ a wide variety of standard character encodings. These facilities should be available to both portable applications and interactive users.

D.1.16 Inter-User Communication

Send messages or transfer files to other users on the same system or other systems on a network. These facilities should be available to both portable applications and interactive users.

D.1.17 System Environment

Display information about the status of the system (activities of users and their interactive and background work, file system utilization, system time, configuration, and presence of optional facilities) and the environment of the user (terminal characteristics, and so on). Inform the system operator/administrator of problems. Control access to user files and other resources.

D.1.18 Printing

Output files on a variety of output device classes, accessing devices on local or network-connected systems. Control (or influence) the formatting, priority scheduling, and output distribution of work. These facilities should be available to both portable applications and interactive users.

D.1.19 Software Development

Develop (create and manage source files, compile/interpret, debug) portable open systems applications and package them for distribution to, and updating of, other systems.

D.2 Portability Capabilities

This section describes the significant portability capabilities of IEEE Std. 1003.1-200x and indicates how the user requirements listed in Section D.1 (on page 3557) are addressed. The capabilities are listed in the same format as the preceding user requirements; they are summarized below:

- Configuration Interrogation
- Process Management
- Access to Data
- Access to the Environment
- Access to Determinism and Performance Enhancements
- Operating System-Dependent Profile
- I/O Interaction
- Internationalization Interaction
- C-Language Extensions
- Command Language
- Interactive Facilities

- Accomplish Multiple Tasks Simultaneously
- Complex Data Manipulation
- File Hierarchy Manipulation
- Locale Configuration
- Inter-User Communication
- System Environment
- Printing
- Software Development

D.2.1 Configuration Interrogation

The *uname*( ) operation provides basic identification of the system. The *sysconf*( ), *pathconf*( ), and *fpathconf*( ) functions and the *getconf* utility provide means to interrogate the implementation to determine how to adapt to the environment in which it is running. These values can be either static (indicating that all instances of the implementation have the same value) or dynamic (indicating that different instances of the implementation have the different values, or that the value may vary for other reasons, such as reconfiguration).

Unsatisfied Requirements

None directly. However, as new areas are added, there will be a need for additional capability in this area.

D.2.2 Process Management

The *fork*( ), *exec* family, and *spawn*( ) functions provide for the creation of new processes or the insertion of new applications into existing processes. The *_Exit*( ), *_exit*( ), *exit*( ), and *abort*( ) functions allow for the termination of a process by itself. The *wait*( ) and *waitpid*( ) functions allow one process to deal with the termination of another.

The *times*( ) function allows for basic measurement of times used by a process. Various functions, including *fstat*( ), *getegid*( ), *geteuid*( ), *getgid*( ), *getgrgid*( ), *getgrnam*( ), *getlogin*( ), *getpid*( ), *getppid*( ), *getpwnam*( ), *getpwuid*( ), *getuid*( ), *lstat*( ), and *stat*( ), provide for access to the identifiers of processes and the identifiers and names of owners of processes (and files).

The various functions operating on environment variables provide for communication of information (primarily user-configurable defaults) from a parent to child processes.

The operations on the current working directory control and interrogate the directory from which relative file name searches start. The *umask*( ) function controls the default protections applied to files created by the process.

The *alarm*( ), *pause*( ), *sleep*( ), *ualarm*( ), and *usleep*( ) operations allow the process to suspend until a timer has expired or to be notified when a period of time has elapsed. The *time*( ) operation interrogates the current time and date.

The signal mechanism provides for communication of events either from other processes or from the environment to the application, and the means for the application to control the effect of these events. The mechanism provides for external termination of a process and for a process to suspend until an event occurs. The mechanism also provides for a value to be associated with an event.

Job control provides a means to group processes and control them as groups, and to control their access to the function between the user and the system (the *controlling terminal*). It also provides the means to suspend and resume processes.

The Process Scheduling option provides control of the scheduling and priority of a process.

The Message Passing option provides a means for interprocess communication involving small amounts of data.

The Memory Management facilities provide control of memory resources and for the sharing of memory. This functionality is mandatory on XSI-conformant systems.

The Threads facilities provide multiple flows of control with a process (threads), synchronization between threads, association of data with threads, and controlled cancelation of threads.

The XSI interprocess communications functionality provide an alternate set of facilities to manipulate semaphores, message queues, and shared memory. These are provided on XSI-conformant systems to support portable applications developed to run on UNIX systems.

D.2.3 Access to Data

The *open*( ), *close*( ), *fclose*( ), *fopen*( ), and *pipe*( ) functions provide for access to files and data. Such files may be regular files, interprocess data channels (pipes), or devices. Additional types of objects in the file system are permitted and are being contemplated for standardization.

The *access*( ), *chmod*( ), *chown*( ), *dup*( ), *dup2*( ), *fchmod*( ), *fcntl*( ), *fstat*( ), *ftruncate*( ), *lstat*( ), *readlink*( ), *realpath*( ), *stat*( ), and *utime*( ) functions allow for control and interrogation of file and file-related objects, (including symbolic links) and their ownership, protections, and timestamps.

The *fgetc*( ), *fputc*( ), *fread*( ), *fseek*( ), *fsetpos*( ), *fwrite*( ), *getc*( ), *getch*( ), *lseek*( ), *putchar*( ), *putc*( ), *read*( ), and *write*( ) functions provide for data transfer from the application to files (in all their forms).

The *closedir*( ), *link*( ), *mkdir*( ), *opendir*( ), *readdir*( ), *rename*( ), *rmdir*( ), *rewinddir*( ), and *unlink*( ) functions provide for a complete set of operations on directories. Directories can arbitrarily contain other directories, and a single file can be mentioned in more than one directory.

The file-locking mechanism provides for advisory locking (protection during transactions) of ranges of bytes (in effect, records) in a file.

The *confstr*( ), *fpathconf*( ), *pathconf*( ), and *sysconf*( ) functions provide for enquiry as to the behavior of the system where variability is permitted.

The Synchronized Input and Output option provides for assured commitment of data to media.

The Asynchronous Input and Output option provides for initiation and control of asynchronous data transfers.

D.2.4 Access to the Environment

The operations and types in the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 11, General Terminal Interface are provided for access to asynchronous serial devices. The primary intended use for these is the controlling terminal for the application (the interaction point between the user and the system). They are general enough to be used to control any asynchronous serial device. The functions are also general enough to be used with many other device types as a user interface when some emulation is provided.

Less detailed access is provided for other device types, but in many instances an application need not know whether an object in the file system is a device or a regular file to operate correctly.

Unsatisfied Requirements

Detailed control of common device classes, specifically magnetic tape, is not provided.

D.2.5 Bounded (Realtime) Response

The Realtime Signals Extension provides queued signals and the prioritization of the handling of signals. The SCHED_FIFO, SCHED_SPORADIC, and SCHED_RR scheduling policies provide control over processor allocation. The Semaphores option provides high-performance synchronization. The Memory Management functions provide memory locking for control of memory allocation, file mapping for high-performance, and shared memory for high-performance interprocess communication. The Message Passing option provides for interprocess communication without being dependent on shared memory.

The Timers option provides a high resolution function called *nanosleep*( ) with a finer resolution than the *sleep*( ) function.

The Typed Memory Objects option, the Monotonic Clock option, and the Timeouts option provide further facilities for applications to use to obtain predictable bounded response.

D.2.6 Operating System-Dependent Profile

IEEE Std. 1003.1-200x makes no distinction between text and binary files. The values of EXIT_SUCCESS and EXIT_FAILURE are further defined.

Unsatisfied Requirements

None known, but the ISO C standard may contain some additional options that could be specified.

D.2.7 I/O Interaction

IEEE Std. 1003.1-200x defines how each of the ISO C standard *stdio* functions interact with the POSIX.1 operations, typically specifying the behavior in terms of POSIX.1 operations.

Unsatisfied Requirements

None.

D.2.8 Internationalization Interaction

The IEEE Std. 1003.1-200x environment operations provide a means to define the environment for *setlocale*( ) and time functions such as *ctime*( ). The *tzset*( ) function is provided to set time conversion information.

The *nl_langinfo*( ) function is provided as an XSI extension to query locale-specific cultural settings.

Unsatisfied Requirements

None.

D.2.9 C-Language Extensions

The *setjmp*( ) and *longjmp*( ) functions are not defined to be cognizant of the signal masks defined for POSIX.1. The *sigsetjmp*( ) and *siglongjmp*( ) functions are provided to fill this gap.

The *_setjmp*( ) and *_longjmp*( ) functions are provided as XSI extensions to support historic practice.

Unsatisfied Requirements

None.

D.2.10 Command Language

The shell command language, as described in Shell and Utilities volume of IEEE Std. 1003.1-200x, Chapter 2, Shell Command Language, is a common language useful in batch scripts, through an API to high-level languages (for the C-Language Binding option, *system*( ) and *popen*( )) and through an interactive terminal (see the *sh* utility). The shell language has many of the characteristics of a high-level language, but it has been designed to be more suitable for user terminal entry and includes interactive debugging facilities. Through the use of pipelining, many complex commands can be constructed from combinations of data filters and other common components. Shell scripts can be created, stored, recalled, and modified by the user with simple editors.

In addition to the basic shell language, the following utilities offer features that simplify and enhance programmatic access to the utilities and provide features normally found only in high-level languages: *basename*, *bc*, *command*, *dirname*, *echo*, *env*, *expr*, *false*, *printf*, *read*, *sleep*, *tee*, *test*, *time**,[2] *true*, *wait*, *xargs*, and all of the special built-in utilities in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.15, Special Built-In Utilities .

Unsatisfied Requirements

None.

D.2.11 Interactive Facilities

The utilities offer a common style of command-line interface through conformance to the Utility Syntax Guidelines (see the Base Definitions volume of IEEE Std. 1003.1-200x, Section 12.2, Utility Syntax Guidelines) and the common utility defaults (see the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 1.11, Utility Description Defaults). The *sh* utility offers an interactive command-line history and editing facility. The following utilities in the User Portability Utilities option have been customized for interactive use: *alias*, *ex*, *fc*, *mailx*, *more*, *talk*, *vi*, *unalias*, and *write*; the *man* utility offers online access to system documentation.

---

2. The utilities listed with an asterisk here and later in this section are present only on systems which support the User Portability Utilities option. There may be further restrictions on the utilities offered with various configuration option combinations; see the individual utility descriptions.

Unsatisfied Requirements

The command line interface to individual utilities is as intuitive and consistent as historical practice allows. Work underway based on graphical user interfaces may be more suitable for novice or occasional users of the system.

D.2.12 Accomplish Multiple Tasks Simultaneously

The shell command language offers background processing through the asynchronous list command form; see the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.9, Shell Commands. The *nohup* utility makes background processing more robust and usable. The *kill* utility can terminate background jobs. When the User Portability Utilities option is supported, the following utilities allow manipulation of jobs: *bg*, *fg*, and *jobs*. Also, if the User Portability Utilities option is supported, the following can support periodic job scheduling, control, and display: *at*, *batch*, *crontab*, *nice*, *ps*, and *renice*.

Unsatisfied Requirements

Terminals with multiple windows may be more suitable for some multi-tasking interactive uses than the job control approach in IEEE Std. 1003.1-200x. See the comments on graphical user interfaces in Section D.2.11 (on page 3564). The *nice* and *renice* utilities do not necessarily take advantage of complex system scheduling algorithms that are supported by the realtime options within IEEE Std. 1003.1-200x.

D.2.13 Complex Data Manipulation

The following utilities address user requirements in this area: *asa*, *awk*, *bc*, *cmp*, *comm*, *csplit**, *cut*, *dd*, *diff*, *ed*, *ex**, *expand**, *expr*, *find*, *fold*, *grep*, *head*, *join*, *od*, *paste*, *pr*, *printf*, *sed*, *sort*, *split**, *tabs**, *tail*, *tr*, *unexpand**, *uniq*, *uudecode**, *uuencode**, and *wc*.

Unsatisfied Requirements

Sophisticated text formatting utilities, such as *troff* or *TeX*, are not included. Standards work in the area of SGML may satisfy this.

D.2.14 File Hierarchy Manipulation

The following utilities address user requirements in this area: *basename*, *cd*, *chgrp*, *chmod*, *chown*, *cksum*, *cp*, *dd*, *df**, *diff*, *dirname*, *du**, *find*, *ls*, *ln*, *mkdir*, *mkfifo*, *mv*, *patch**, *pathchk*, *pax*, *pwd*, *rm*, *rmdir*, *test*, and *touch*.

Unsatisfied Requirements

Some graphical user interfaces offer more intuitive file manager components that allow file manipulation through the use of icons for novice users.

D.2.15 Locale Configuration

The standard utilities are affected by the various *LC_* variables to achieve locale-dependent operation: character classification, collation sequences, regular expressions and shell pattern matching, date and time formats, numeric formatting, and monetary formatting. When the POSIX2_LOCALEDEF option is supported, applications can provide their own locale definition files. The following utilities address user requirements in this area: *date*, *ed*, *ex**, *find*, *grep*, *locale*, *localedef*, *more**, *sed*, *sh*, *sort*, *tr*, *uniq*, and *vi**.

The *iconv*( ), *iconv_close*( ), and *iconv_open*( ) functions are available to allow an application to convert character data between supported character sets.

The *gencat* utility and the *catopen*( ), *catclose*( ), and *catgets*( ) functions for message catalog manipulation are available on XSI-conformant systems.

Unsatisfied Requirements

Some aspects of multi-byte character and state-encoded character encodings have not yet been addressed. The C-language functions, such as *getopt*( ), are generally limited to single-byte characters. The effect of the *LC_MESSAGES* variable on message formats is only suggested at this time.

D.2.16 Inter-User Communication

The following utilities address user requirements in this area: *cksum*, *mailx**, *mesg**, *patch**, *pax*, *talk**, *uudecode**, *uuencode**, *who**, and *write**.

The historical UUCP utilities are included on XSI-conformant systems.

Unsatisfied Requirements

None.

D.2.17 System Environment

The following utilities address user requirements in this area: *chgrp*, *chmod*, *chown*, *df**, *du**, *env*, *getconf*, *id*, *logger*, *logname*, *mesg**, *newgrp**, *ps**, *stty*, *tput**, *tty*, *umask*, *uname*, and *who**.

The *closelog*( ), *openlog*( ), *setlogmask*( ), and *syslog*( ) functions provide System Logging facilities on XSI-conformant systems; these are analogous to the *logger* utility.

Unsatisfied Requirements

None.

D.2.18 Printing

The following utilities address user requirements in this area: *pr* and *lp*.

Unsatisfied Requirements

There are no features to control the formatting or scheduling of the print jobs.

D.2.19 Software Development

The following utilities address user requirements in this area: *ar*, *asa*, *awk*, *c99*, *ctags**, *fort77*, *getconf*, *getopts*, *lex*, *localedef*, *make*, *nm**, *od*, *patch**, *pax*, *strings**, *strip*, *time**, and *yacc*.

The *system*( ), *popen*( ), *pclose*( ), *regcomp*( ), *regexec*( ), *regerror*( ), *regfree*( ), *fnmatch*( ), *getopt*( ), *glob*( ), *globfree*( ), *wordexp*( ), and *wordfree*( ) functions allow C-language programmers to access some of the interfaces used by the utilities, such as argument processing, regular expressions, and pattern matching.

The SCCS source-code control system utilities are available on systems supporting the XSI Development option.

Unsatisfied Requirements

There are no language-specific development tools related to languages other than C and FORTRAN. The C tools are more complete and varied than the FORTRAN tools. There is no data dictionary or other CASE-like development tools.

D.2.20 Future Growth

It is arguable whether or not all functionality to support applications is potentially within the scope of IEEE Std. 1003.1-200x. As a simple matter of practicality, it cannot be. Areas such as general networking, graphics, application domain-specific functionality, windowing, and so on, should be in unique standards. As such, they are properly ‘‘Unsatisfied Requirements’’ in terms of providing fully portable applications, but ones which are outside the scope of IEEE Std. 1003.1-200x.

D.3 Profiling Considerations

This section offers guidance to writers of profiles on how the configurable options, limits, and optional behavior of IEEE Std. 1003.1-200x should be cited in profiles. Profile writers should consult the general guidance in POSIX.0 when writing POSIX Standardized Profiles.

The information in this section is an inclusive list of features that should be considered by profile writers. Further subsetting of IEEE Std. 1003.1-200x, including the specification of behavior currently described as unspecified, undefined, implementation-defined, or with the verbs ‘‘may’’ or ‘‘need not’’ violates the intent of the developers of IEEE Std. 1003.1-200x and the guidelines of ISO/IEC TR 10000-1. A set of profiling option groups is described in the Base Definitions volume of IEEE Std. 1003.1-200x, based on the IEEE Std. 1003.13-1998 options, with the addition of a new profiling option group called _POSIX_NETWORKING.

D.3.1 Configuration Options

There are two set of options suggested by IEEE Std. 1003.1-200x: those for POSIX-conforming systems and those for X/Open System Interface (XSI) conformance. The requirements for XSI conformance are documented in the Base Definitions volume of IEEE Std. 1003.1-200x and not discussed further here, as they superset the POSIX conformance requirements.

D.3.2 Configuration Options (Shell and Utilities)

There are three broad optional configurations for the Shell and Utilities volume of IEEE Std. 1003.1-200x: basic execution system, development system, and user portability interactive system. The options to support these, and other minor configuration options, are listed in the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 2, Conformance. Profile writers should consult the following list and the comments concerning user requirements addressed by various components in Section D.2 (on page 3560).

POSIX2_UPE

The system supports the User Portability Utilities option.

This option is a requirement for a user portability interactive system. It is required frequently except for those systems, such as embedded realtime or dedicated application systems, that support little or no interactive time-sharing work by users or operators. XSI-conformant systems support this option.

POSIX2_SW_DEV

The system supports the Software Development Utilities option.

This option is required by many systems, even those in which actual software development does not occur. The *make* utility, in particular, is required by many application software packages as they are installed onto the system. If POSIX2_C_DEV is supported, POSIX2_SW_DEV is almost a mandatory requirement because of *ar* and *make*.

POSIX2_C_BIND

The system supports the C-Language Bindings option.

This option is required on some systems developing complex C applications or on any system installing C applications in source form that require the functions in this option. The *system*( ) and *popen*( ) functions, in particular, are widely used by applications; the others are rather more specialized.

POSIX2_C_DEV

The system supports the C-Language Development Utilities option.

This option is required by many systems, even those in which actual C-language software development does not occur. The *c99* utility, in particular, is required by many application software packages as they are installed onto the system. The *lex* and *yacc* utilities are used less frequently.

POSIX2_FORT_DEV

The system supports the FORTRAN Development Utilities option

As with C, this option is needed on any system developing or installing FORTRAN applications in source form.

POSIX2_FORT_RUN

The system supports the FORTRAN Runtime Utilities option.

This option is required for some FORTRAN applications that need the *asa* utility to convert Hollerith printing statement output. It is unknown how frequently this occurs.

POSIX2_LOCALEDEF

The system supports the creation of locales.

This option is needed if applications require their own customized locale definitions to operate. It is presently unknown whether many applications are dependent on this. However, the option is virtually mandatory for systems in which internationalized applications are developed.

XSI-conformant systems support this option.

POSIX2_PBS

The system supports the Batch Environment option.

POSIX2_PBS_ACCOUNTING

The system supports the optional feature of accounting within the Batch Environment option. It will be required in servers that implement the optional feature of accounting.

POSIX2_PBS_CHECKPOINT

The systems supports the optional feature of checkpoint/restart within the Batch Environment option.

POSIX2_PBS_LOCATE

The system supports the optional feature of locating batch jobs within the Batch Environment option.

POSIX2_PBS_MESSAGE

The system supports the optional feature of sending messages to batch jobs within the Batch Environment option.

POSIX2_PBS_TRACK

The system supports the optional feature of tracking batch jobs within the Batch Environment option.

POSIX2_CHAR_TERM

The system supports at least one terminal type capable of all operations described in IEEE Std. 1003.1-200x.

On systems with POSIX2_UPE, this option is almost always required. It was developed solely to allow certain specialized vendors and user applications to bypass the requirement for general-purpose asynchronous terminal support. For example, an application and system that was suitable for block-mode terminals, such as IBM 3270s, would not need this option.

XSI-conformant systems support this option.

D.3.3 Configurable Limits

Very few of the limits need to be increased for profiles. No profile can cite lower values.

{POSIX2_BC_BASE_MAX}
{POSIX2_BC_DIM_MAX}
{POSIX2_BC_SCALE_MAX}
{POSIX2_BC_STRING_MAX}

No increase is anticipated for any of these *bc* values, except for very specialized applications involving huge numbers.

{POSIX2_COLL_WEIGHTS_MAX}

Some natural languages with complex collation requirements require an increase from the default 2 to 4; no higher numbers are anticipated.

{POSIX2_EXPR_NEST_MAX}
No increase is anticipated.

{POSIX2_LINE_MAX}
This number is much larger than most historical applications have been able to use. At some future time, applications may be rewritten to take advantage of even larger values.

{POSIX2_RE_DUP_MAX}
No increase is anticipated.

{POSIX2_VERSION}
This is actually not a limit, but a standard version stamp. Generally, a profile should specify Shell and Utilities volume of IEEE Std. 1003.1-200x, Chapter 2, Shell Command Language by name in the normative references section, not this value.

D.3.4 Configuration Options (System Interfaces)

{NGROUPS_MAX}
A non-zero value indicates that the implementation supports supplementary groups.

This option is needed where there is a large amount of shared use of files, but where a certain amount of protection is needed. Many profiles[3] are known to require this option; it should only be required if needed, but it should never be prohibited.

_POSIX_ADVISORY_INFO
The system provides advisory information for file management.

This option allows the application to specify advisory information that can be used to achieve better or even deterministic response time in file manager or input and output operations.

_POSIX_ASYNCHRONOUS_IO
The system provides concurrent process execution and input and output transfers.

This option was created to support historical systems that did not provide the feature. It should only be required if needed, but it should never be prohibited.

_POSIX_BARRIERS
The system supports barrier synchronization.

This option was created to allow efficient synchronization of multiple parallel threads in multi-processor systems in which the operation is supported in part by the hardware architecture.

_POSIX_CHOWN_RESTRICTED
The system restricts the right to ''give away'' files to other users.

This option should be carefully investigated before it is required. Some applications expect that they can change the ownership of files in this way. It is provided where either security or system account requirements cause this ability to be a problem. It is also known to be specified in many profiles.

---

3. There are no formally approved profiles of IEEE Std. 1003.1-200x at the time of publication; the reference here is to various profiles generated by private bodies or governments.

_POSIX_CLOCK_SELECTION

The system supports the Clock Selection option.

This option allows applications to request a high resolution sleep in order to suspend a thread during a relative time interval, or until an absolute time value, using the desired clock. It also allows the application to select the clock used in a *pthread_cond_timedwait*( ) function call.

_POSIX_CPUTIME

The system supports the Process CPU-Time Clocks option.

This option allows applications to use a new clock that measures the execution times of processes or threads, and the possibility to create timers based upon these clocks, for runtime detection (and treatment) of execution time overruns.

_POSIX_FSYNC

The system supports file synchronization requests.

This option was created to support historical systems that did not provide the feature. Applications that are expecting guaranteed completion of their input and output operations should require the _POSIX_SYNC_IO option. This option should never be prohibited.

XSI-conformant systems support this option.

_POSIX_IPV6

The system supports facilities related to Internet Protocol Version 6 (IPv6).

This option was created to allow systems to transition to IPv6.

_POSIX_JOB_CONTROL

Job control facilities are mandatory in IEEE Std. 1003.1-200x.

The option was created primarily to support historical systems that did not provide the feature. Many existing profiles now require it; it should only be required if needed, but it should never be prohibited. Most applications that use it can run when it is not present, although with a degraded level of user convenience.

_POSIX_MAPPED_FILES

The system supports the mapping of regular files into the process address space.

XSI-conformant systems support this option.

Both this option and the Shared Memory Objects option provide shared access to memory objects in the process address space. The functions defined under this option provide the functionality of existing practice for mapping regular files. This functionality was deemed unnecessary, if not inappropriate, for embedded systems applications and, hence, is provided under this option. It should only be required if needed, but it should never be prohibited.

_POSIX_MEMLOCK

The system supports the locking of the address space.

This option was created to support historical systems that did not provide the feature. It should only be required if needed, but it should never be prohibited.

_POSIX_MEMLOCK_RANGE

The system supports the locking of specific ranges of the address space.

For applications that have well-defined sections that need to be locked and others that do not, IEEE Std. 1003.1-200x supports an optional set of functions to lock or unlock a range of process addresses. The following are two reasons for having a means to lock down a specific range:

1. An asynchronous event handler function that must respond to external events in a deterministic manner such that page faults cannot be tolerated

2. An input/output ''buffer'' area that is the target for direct-to-process I/O, and the overhead of implicit locking and unlocking for each I/O call cannot be tolerated

It should only be required if needed, but it should never be prohibited.

_POSIX_MEMORY_PROTECTION
The system supports memory protection.

XSI-conformant systems support this option.

The provision of this option typically imposes additional hardware requirements. It should never be prohibited.

_POSIX_PRIORITIZED_IO
The system provides prioritization for input and output operations.

The use of this option may interfere with the ability of the system to optimize input and output throughput. It should only be required if needed, but it should never be prohibited.

_POSIX_MESSAGE_PASSING
The system supports the passing of messages between processes.

This option was created to support historical systems that did not provide the feature. The functionality adds a high-performance XSI interprocess communication facility for local communication. It should only be required if needed, but it should never be prohibited.

_POSIX_MONOTONIC_CLOCK
The system supports the Monotonic Clock option.

This option allows realtime applications to rely on a monotonically increasing clock that does not jump backwards, and whose value does not change except for the regular ticking of the clock.

_POSIX_PRIORITY_SCHEDULING
The system provides priority-based process scheduling.

Support of this option provides predictable scheduling behavior, allowing applications to determine the order in which processes that are ready to run are granted access to a processor. It should only be required if needed, but it should never be prohibited.

_POSIX_REALTIME_SIGNALS
The system provides prioritized, queued signals with associated data values.

This option was created to support historical systems that did not provide the features. It should only be required if needed, but it should never be prohibited.

_POSIX_REGEXP
Support for regular expression facilities are mandatory in IEEE Std. 1003.1-200x.

_POSIX_SAVED_IDS
Support for this feature is mandatory in IEEE Std. 1003.1-200x.

Certain classes of applications rely on it for proper operation, and there is no alternative short of giving the application root privileges on most implementations that did not provide _POSIX_SAVED_IDS.

_POSIX_SEMAPHORES

The system provides counting semaphores.

This option was created to support historical systems that did not provide the feature. It should only be required if needed, but it should never be prohibited.

_POSIX_SHARED_MEMORY_OBJECTS

The system supports the mapping of shared memory objects into the process address space.

Both this option and the Memory Mapped Files option provide shared access to memory objects in the process address space. The functions defined under this option provide the functionality of existing practice for shared memory objects. This functionality was deemed appropriate for embedded systems applications and, hence, is provided under this option. It should only be required if needed, but it should never be prohibited.

_POSIX_SHELL

Support for the *sh* utility command line interpreter is mandatory in IEEE Std. 1003.1-200x.

_POSIX_SPAWN

The system supports the spawn option.

This option provides applications with an efficient mechanism to spawn execution of a new process.

_POSIX_SPINLOCKS

The system supports spin locks.

This option was created to support a simple and efficient synchronization mechanism for threads executing in multi-processor systems.

_POSIX_SPORADIC_SERVER

The system supports the sporadic server scheduling policy.

This option provides applications with a new scheduling policy for scheduling aperiodic processes or threads in hard realtime applications.

_POSIX_SYNCHRONIZED_IO

The system supports guaranteed file synchronization.

This option was created to support historical systems that did not provide the feature. Applications that are expecting guaranteed completion of their input and output operations should require this option, rather than the File Synchronization option. It should only be required if needed, but it should never be prohibited.

_POSIX_THREADS

The system supports multiple threads of control within a single process.

This option was created to support historical systems that did not provide the feature. Applications written assuming a multi-threaded environment would be expected to require this option. It should only be required if needed, but it should never be prohibited.

XSI-conformant systems support this option.

_POSIX_THREAD_ATTR_STACKADDR

The system supports specification of the stack address for a created thread.

Applications may take advantage of support of this option for performance benefits, but dependence on this feature should be minimized. This option should never be prohibited.

XSI-conformant systems support this option.

_POSIX_THREAD_ATTR_STACKSIZE

The system supports specification of the stack size for a created thread.

Applications may require this option in order to ensure proper execution, but such usage limits portability and dependence on this feature should be minimized. It should only be required if needed, but it should never be prohibited.

XSI-conformant systems support this option.

_POSIX_THREAD_PRIORITY_SCHEDULING

The system provides priority-based thread scheduling.

Support of this option provides predictable scheduling behavior, allowing applications to determine the order in which threads that are ready to run are granted access to a processor. It should only be required if needed, but it should never be prohibited.

_POSIX_THREAD_PRIO_INHERIT

The system provides mutual exclusion operations with priority inheritance.

Support of this option provides predictable scheduling behavior, allowing applications to determine the order in which threads that are ready to run are granted access to a processor. It should only be required if needed, but it should never be prohibited.

_POSIX_THREAD_PRIO_PROTECT

The system supports a priority ceiling emulation protocol for mutual exclusion operations.

Support of this option provides predictable scheduling behavior, allowing applications to determine the order in which threads that are ready to run are granted access to a processor. It should only be required if needed, but it should never be prohibited.

_POSIX_THREAD_PROCESS_SHARED

The system provides shared access among multiple processes to synchronization objects.

This option was created to support historical systems that did not provide the feature. It should only be required if needed, but it should never be prohibited.

XSI-conformant systems support this option.

_POSIX_THREAD_SAFE_FUNCTIONS

The system provides thread-safe versions of all of the POSIX.1 functions.

This option is required if the Threads option is supported. This is a separate option because thread-safe functions are useful in implementations providing other mechanisms for concurrency. It should only be required if needed, but it should never be prohibited.

XSI-conformant systems support this option.

_POSIX_THREAD_SPORADIC_SERVER

The system supports the thread sporadic server scheduling policy.

Support for this option provides applications with a new scheduling policy for scheduling aperiodic threads in hard realtime applications.

_POSIX_TIMEOUTS

The system provides timeouts for some blocking services.

This option was created to provide a timeout capability to system services, thus allowing applications to include better error detection, and recovery capabilities.

_POSIX_TIMERS

The system provides higher resolution clocks with multiple timers per process.

This option was created to support historical systems that did not provide the features. This option is appropriate for applications requiring higher resolution timestamps or needing to control the timing of multiple activities. It should only be required if needed, but it should never be prohibited.

_POSIX_TRACE

The system supports the trace option.

This option was created to allow applications to perform tracing.

_POSIX_TRACE_EVENT_FILTER

The system supports the trace event filter option.

This option is dependent on support of the Trace option.

_POSIX_TRACE_INHERIT

The system supports the trace inherit option.

This option is dependent on support of the Trace option.

_POSIX_TRACE_LOG

The system supports the trace log option.

This option is dependent on support of the Trace option.

_POSIX_TYPED_MEMORY_OBJECTS

The system supports typed memory objects.

This option was created to allow realtime applications to access different kinds of physical memory, and allow processes in these applications to share portions of this memory.

D.3.5 Configurable Limits

In general, the configurable limits in the **<limits.h>** header defined in the Base Definitions volume of IEEE Std. 1003.1-200x have been set to minimal values; many applications or implementations may require larger values. No profile can cite lower values.

{AIO_LISTIO_MAX}

The current minimum is likely to be inadequate for most applications. It is expected that this value will be increased by profiles requiring support for list input and output operations.

{AIO_MAX}

The current minimum is likely to be inadequate for most applications. It is expected that this value will be increased by profiles requiring support for asynchronous input and output operations.

{AIO_PRIO_DELTA_MAX}

The functionality associated with this limit is needed only by sophisticated applications. It is not expected that this limit would need to be increased under a general-purpose profile.

{ARG_MAX}

The current minimum is likely to need to be increased for profiles, particularly as larger amounts of information are passed through the environment. Many implementations are believed to support larger values.

{CHILD_MAX}

The current minimum is suitable only for systems where a single user is not running applications in parallel. It is significantly too low for any system also requiring windows, and if _POSIX_JOB_CONTROL is specified, it should be raised.

{CLOCKRES_MIN}

It is expected that profiles will require a finer granularity clock, perhaps as fine as 1 µs, represented by a value of 1 000 for this limit.

{DELAYTIMER_MAX}

It is believed that most implementations will provide larger values.

{LINK_MAX}

For most applications and usage, the current minimum is adequate. Many implementations have a much larger value, but this should not be used as a basis for raising the value unless the applications to be used require it.

{LOGIN_NAME_MAX}

This is not actually a limit, but an implementation parameter. No profile should impose a requirement on this value.

{MAX_CANON}

For most purposes, the current minimum is adequate. Unless high-speed burst serial devices are used, it should be left as is.

{MAX_INPUT}

See {MAX_CANON}.

{MQ_OPEN_MAX}

The current minimum should be adequate for most profiles.

{MQ_PRIO_MAX}

The current minimum corresponds to the required number of process scheduling priorities. Many realtime practitioners believe that the number of message priority levels ought to be the same as the number of execution scheduling priorities.

{NAME_MAX}

Many implementations now support larger values, and many applications and users assume that larger names can be used. Many existing profiles also specify a larger value. Specifying this value will reduce the number of conforming implementations, although this might not be a significant consideration over time. Values greater than 255 should not be required.

{NGROUPS_MAX}

The value selected will typically be 8 or larger.

{OPEN_MAX}

The historically common value for this has been 20. Many implementations support larger values. If applications that use larger values are anticipated, an appropriate value should be specified.

{PAGESIZE}

This is not actually a limit, but an implementation parameter. No profile should impose a requirement on this value.

{PATH_MAX}

Historically, the minimum has been either 1 024 or indefinite, depending on the implementation. Few applications actually require values larger than 256, but some users may create file hierarchies that must be accessed with longer paths. This value should only be changed if there is a clear requirement.

{PIPE_BUF}

The current minimum is adequate for most applications. Historically, it has been larger. If applications that write single transactions larger than this are anticipated, it should be increased. Applications that write lines of text larger than this probably do not need it increased, as the text line is delimited by a newline.

{POSIX_VERSION}
This is actually not a limit, but a standard version stamp. Generally, a profile should specify IEEE Std. 1003.1-200x by a name in the normative references section, not this value.

{PTHREAD_DESTRUCTOR_ITERATIONS}
It is unlikely that applications will need larger values to avoid loss of memory resources.

{PTHREAD_KEYS_MAX}
The current value should be adequate for most profiles.

{PTHREAD_STACK_MIN}
This should not be treated as an actual limit, but as an implementation parameter. No profile should impose a requirement on this value.

{PTHREAD_THREADS_MAX}
It is believed that most implementations will provide larger values.

{RTSIG_MAX}
The current limit was chosen so that the set of POSIX.1 signal numbers can fit within a 32-bit field. It is recognized that most existing implementations define many more signals than are specified in POSIX.1 and, in fact, many implementations have already exceeded 32 signals (including the ‘‘null signal’’). Support of {_POSIX_RTSIG_MAX} additional signals may push some implementations over the single 32-bit word line, but is unlikely to push any implementations that are already over that line beyond the 64 signal line.

{SEM_NSEMS_MAX}
The current value should be adequate for most profiles.

{SEM_VALUE_MAX}
The current value should be adequate for most profiles.

{SSIZE_MAX}
This limit reflects fundamental hardware characteristics (the size of an integer), and should not be specified unless it is clearly required. Extreme care should be taken to assure that any value that might be specified does not unnecessarily eliminate implementations because of accidents of hardware design.

{STREAM_MAX}
This limit is very closely related to {OPEN_MAX}. It should never be larger than {OPEN_MAX}, but could reasonably be smaller for application areas where most files are not accessed through *stdio*. Some implementations may limit {STREAM_MAX} to 20 but allow {OPEN_MAX} to be considerably larger. Such implementations should be allowed for if the applications permit.

{TIMER_MAX}
The current limit should be adequate for most profiles, but it may need to be larger for applications with a large number of asynchronous operations.

{TTY_NAME_MAX}
This is not actually a limit, but an implementation parameter. No profile should impose a requirement on this value.

{TZNAME_MAX}
The minimum has been historically adequate, but if longer timezone names are anticipated (particularly such values as UTC–1), this should be increased.

D.3.6 Optional Behavior

In IEEE Std. 1003.1-200x, there are no instances of the terms unspecified, undefined, implementation-defined, or with the verbs ''may'' or ''need not'', that the developers of IEEE Std. 1003.1-200x anticipate or sanction as suitable for profile or test method citation. All of these are merely warnings to portable applications to avoid certain areas that can vary from system to system, and even over time on the same system. In many cases, these terms are used explicitly to support extensions, but profiles should not anticipate and require such extensions; future versions of IEEE Std. 1003.1-200x may do so.