Adopted for Use by the Federal Government

FEDERAL INFORMATION PROCESSING STANDARDS
FIPS

FIPS PUB 151

See Notice on Inside Front Cover

IEEE P1003.1/Draft 12
October 12, 1987

Portable Operating System Interface for Computer Environments

Sponsor

Technical Committee on Operating Systems of the IEEE Computer Society

The Institute of Electrical and Electronics Engineers, Inc
345 East 47th Street, New York, NY 10017, USA

Portable Operating System Interface for Computer Environments

Sponsor

Technical Committee on Operating Systems
of the
IEEE
Computer Society

P1003.1 / DRAFT 12
October 12, 1987

This standard has been adopted for Federal Government use.

Details concerning its use within the Federal Government are contained in Federal Information Processing Standards Publication 151, POSIX: Portable Operating System Interface for Computer Environments. For a complete list of publications available in the Federal Information Processing Standards Series, write to the Standards Processing Coordinator (ADP), National Computer and Telecommunications Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899.

USER NOTE: Draft 12 of IEEE 1003.1 is not the most current version of this standard and is not identical to IEEE Std 1003.1-1988. IEEE Std 1003.1-1988 is the published version of Draft 13, which was approved by the IEEE Standards Board on August 22, 1988.

Foreword

(This Foreword is not a part of IEEE Std 1003.1, IEEE Standard Portable Operating System Interface for Computer Environments.)

The purpose of this standard is to define a standard operating system interface and environment based on the UNIX* Operating System documentation to support application portability at the source level. This is intended for systems implementors and applications software implementors.

In its present form, the standard focuses primarily on the C Language interface to the operating system.

IEEE Std 1003.1 is the first of a group of proposed standards known colloquially, and collectively, as POSIX†. The other POSIX standards are described in Appendix A.

Organization of the Standard

The standard is divided into four parts:

- Statement of scope (Chapter 1)
- Definitions and global concepts (Chapter 2)
- The various interface facilities (Chapters 3 through 9)
- Data interchange format (Chapter 10)

This foreword and the appendices are not considered part of the standard.

Most of the sections describe a single service interface. The C Language binding for the service interface is given in the subsection labeled **Synopsis**. The **Description** subsection provides a specification of the operation performed by the service interface. Some examples may be provided to illustrate the interfaces described. In most cases there are also **Returns** and **Errors** subsections specifying return values and possible error conditions. **References** are used to direct the reader to other related sections. Additional material to complement sections in the standard may be found in **Rationale and Notes**, Appendix B. This appendix provides historical perspectives into the technical choices made by the 1003.1 Working Group. It also provides information to emphasize consequences of the interfaces described in the corresponding section of the standard.

---

* UNIX is a registered trademark of AT&T.

† POSIX is pronounced *pahz-icks*, similar to *positive*.

In publishing this standard, both the IEEE and the 1003.1 Working Group simply intend to provide a yardstick against which various operating system implementations can be measured for conformance. It is *not* the intent of either the IEEE or the 1003.1 Working Group to measure or rate any products, to reward or sanction any vendors of products for conformance or lack of conformance to this standard, or to attempt to enforce this standard by these or any other means. The responsibility for determining the degree of conformance or lack thereof with this standard rests solely with the individual who is evaluating the product claiming to be in conformance with the standard. (See **Verification Testing** §A.2.3 for additional information on this subject.)

Base Documents

The various interface facilities described herein are based on the *1984 /usr/group Standard* derived and published by the /usr/group Standards Committee, Santa Clara, California. The *1984 /usr/group Standard,* and subsequent work of the 1003.1 Working Group is largely based on UNIX Seventh Edition, System III, System V, 4.2BSD, and 4.3BSD documentation, but wherever possible, compatibility with other UNIX-derived systems and compatible systems has been maintained.

The IEEE is grateful to both AT&T and /usr/group for permission to use their materials.

Extensions and Supplements to this Standard

Activities to extend this standard to address additional requirements are in progress and similar efforts can be anticipated in the future. This is an outline of how these extensions will be incorporated, and also how users of this document can keep track of that status.

Extensions are approved as "Supplements" to this document, following the IEEE Standards Procedures.

Approved Supplements are published separately and distributed with orders from the IEEE for this document until the full document is reprinted and such supplements are incorporated in their proper positions.

If you have any question about the completeness of your version, you may contact the IEEE Computer Society (*phone # to be provided*) or the IEEE Standards Office (*phone # to be provided*) to determine what supplements have been published. Published supplements will be available for a modest fee.

Supplements are numbered in the same format as the main document, and with unique positions as either subsections or main sections. A supplement may include new subsections in various sections of the main document as well as new main sections. Supplements may include new sections in already approved supplements. However, the overall numbering shall be unique so that two supplements do not use the same numbers unless one replaces the other.

Supplements may contain either required functions or optional facilities. Supplements may add additional conformance requirements (see **Conformance** §2.2) defining new classes of conforming systems or applications.

It is desirable, but perhaps not avoidable, that supplements do not change the functionality of the already defined facilities.

Supplements are not used to provide a general update of the standard. This is done through the review procedure as specified by the IEEE.

The following areas are under active consideration at this time, or are expected to become active in the near future.

- Shell and Utility facilities — P1003.2 (see **Shell and Utilities** §A.2.2);
- Verification Testing — P1003.3 (see **Verification Testing** §A.2.3);
- Real Time facilities — P1003.4 (see **Real Time Extensions** §A.2.4);
- Secure/Trusted System considerations;
- FORTRAN Language bindings;
- Ada* Language bindings;
- Language-independent service descriptions;
- An overall guide to POSIX-based or related Open Systems standards.

(See Appendix A for additional information.) If you have interest in participating in the working groups addressing these issues, please send your name, address, and phone number to the:

Secretary, IEEE Standards Board
Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street
New York, NY 10017

and ask to have this forwarded to the chairperson of the appropriate P1003 working group.

* Ada is a trademark of the Department of Defense.

Editor's Notes

This section will not appear in the final document. It is used for editorial comments concerning Draft 12.

This draft uses small numbers in the right margin in lieu of change bars. ''8'' denotes changes from Draft 7 (the Trial Use Standard) to Draft 8. ''9'' denotes changes from Draft 8 to Draft 9. ''A'' denotes changes from Draft 9 to Draft 10 (in hex). ''B'' denotes changes from Draft 10 to Draft 11 (in hex). ''C'' denotes changes from Draft 11 to Draft 12 (in hex). Deleted text uses the same symbols, but will generally be noted by a blank line containing only the change symbol. It should be noted that, due to the algorithms used by troff, some change symbols are overlaid by a following change on the same line, and are therefore obscured. For the future, we will continue hexadecimally and hope that Full Use is achieved before Draft 16. The Full Use standard will have neither change marks or line numbers. The correctness or format of these symbols are not ballotable issues.

All of the header paragraphs in the Errors *sections have changed slightly (''shall return −1'' replaces ''shall fail''); these changes are not marked.*

Please report typographical errors and editorial changes directly to:

Hal Jespersen
UniSoft Corporation
6121 Hollis Street
Emeryville, CA 94608-2092
(415) 420-6448
UUCP: {uunet,amdahl,sun}!unisoft!hlj

(Electronic mail is preferred.)

IEEE Std 1003.1 was prepared by the 1003.1 Working Group, sponsored by the Technical Committee on Operating Systems of the IEEE Computer Society.

At the time this standard was approved, the membership of the 1003.1 Working Group was as follows:

Editor's Note: This list will be included in the final printed standard.

Steering Committee B

Joseph Boykin	*Chair, TCOS*	C
James Isaak	*Chair, P1003.1*	C
Hal Jespersen	*Technical Editor, P1003.1*	C
Shane P. McCarron	*Secretary, P1003.1*	C
Donn S. Terry	*Co-Chair, P1003.1*	C

Working Group

Name	*Name*	*Name*
Name	*Name*	*Name*
Name	*Name*	*Name*

The following persons were members of the 1003.1 Balloting Group that approved the standard for submission to the IEEE Standards Board:

Heinz Lycklama	*/usr/group Institutional Representative*	C
Michael Lambert	*X/OPEN Institutional Representative*	C
John S. Quarterman	*USENIX Institutional Representative*	C

Editor's Note: This list will be included in the final printed standard.

Name	*Name*	*Name*
Name	*Name*	*Name*
Name	*Name*	*Name*

Contents

SECTION PAGE

SECTION PAGE

APPENDICES

SECTION PAGE

SECTION PAGE

SECTION PAGE

Portable Operating System Interface for Computer Environments c

1. Scope

This standard defines a standard operating system interface and environment to support 8
application portability at the source code level. It is intended to be used by both application developers and system implementors.

Initially, the focus of the standard will be on the C language interface. In future c
revisions, this will be divided into several parts. The first part will provide a functional c
definition of the service interfaces. The following parts will specify the binding between c
these service interfaces and specific programming languages, with the second part c
describing the C language binding. c

This effort entails four major components: c

1. Definitions for terminology and objects referred to in the standard (in the 8
   case of objects, their structure, operations that modify objects, and the effects of these operations);

2. System service interfaces and subroutines; c

3. C language binding for the system services; c

4. Interface issues, including portability, error handling, and recovery. 9

The following areas are outside of the scope of this standard: 8

- User interface (shell) and associated commands
- Network protocols
- Graphics interfaces
- Data base management system interfaces
- Record I/O considerations

- Object or binary code portability

(See Appendix A for information about ongoing efforts in some of these areas.)

This standard describes the external characteristics and facilities that are of importance to applications developers, rather than on the internal construction techniques employed to achieve these capabilities. Special emphasis is placed on those functions and facilities that are needed in a wide variety of commercial applications.

This standard has been defined exclusively at the source code level. The objective is that a Strictly Conforming Application source program can be compiled to execute on a conforming implementation.

2. Definitions and General Requirements

2.1 Terminology

The following terms are used in this standard:

implementation defined

A value or behavior is **implementation defined** if the implementation defines and documents the requirements for correct program construct and correct data.

may

With respect to implementations, the word **may** is to be interpreted as an optional feature that is not required in this standard and can be provided. With respect to **Strictly Conforming Applications**, the word **may** means that the optional feature shall not be used.

shall

In this standard, the word **shall** is to be interpreted as a requirement on the implementation or on **Strictly Conforming Applications**, where appropriate.

should

With respect to implementations, the word **should** is to be interpreted as an implementation recommendation, but not a requirement. With respect to applications, the word **should** is to be interpreted as recommended programming practice for applications and a requirement for **Strictly Conforming Applications**.

undefined

A value or behavior is **undefined** if the standard imposes no portability requirements for erroneous program construct, erroneous data, or use of an indeterminate value.

unspecified

A value or behavior is **unspecified** if the standard imposes no portability requirements for a correct program construct or correct data.

2.2 Conformance

2.2.1 Implementation Conformance

2.2.1.1 Requirements

A **conforming implementation** shall meet all of the following criteria:

> The system shall support all required interfaces defined within this standard. These interfaces shall support the functional behavior described herein.

> The system may provide additional functions or facilities not required by this standard. Nonstandard extensions should be identified as such in the system documentation. Nonstandard extensions, when used, may change the behavior of functions or facilities defined by this standard. In such cases, the system documentation shall define an environment in which an application can be run with the behavior specified by the standard. In no case shall such an environment require modification of a **Strictly Conforming Application**.

2.2.1.2 Documentation

A document with the following information shall be available for an implementation claiming conformance to IEEE Std 1003.1. This document shall have the same structure as this standard, with the information presented in the appropriately numbered sections. The document shall not contain information about extended facilities or capabilities outside the scope of this standard.

The document shall contain a conformance statement that indicates the full name, number, and date of the standard that applies. The conformance section may also list software standards approved by ISO or any ISO member body that are available for use by a Conforming Application. Applicable characteristics where documentation is required by one of these standards, or by standards of government bodies, may also be included.

The document shall describe the contents of the **<limits.h>** and **<unistd.h>** headers, stating values, the conditions under which those values may change, and the limits of such variations.

The document should describe the nature of the implementation for all implementation defined features identified in this standard.

The document should specify the behavior of the implementation in those sections of this standard where it is stated that implementations may vary.

2.2.2 Application Conformance

All applications claiming conformance to this standard shall use only **Conforming Languages** §2.2.3, and shall fall within one of the following categories:

2.2.2.1 Strictly Conforming Application

A **Strictly Conforming Application** is an application that requires only the facilities described in this standard and the applicable language standards. Such an application shall accept any behavior described in this standard as implementation defined, and for symbolic constants, shall accept any value in the range permitted by this standard. Such applications are permitted to adapt to the availability of facilities whose availability is indicated by the constants in **<limits.h>** §2.9 and **<unistd.h>** §2.10.

2.2.2.2 Conforming Application

A **Conforming Application** is an application that uses only the facilities described in this standard and approved **Conforming Language** bindings for any ANSI standard. Such an application shall include in its statement of conformance all options and limit dependencies, all other ANSI standards used, and any other applications required.

2.2.2.3 Conforming Application Using Extensions

A **Conforming Application Using Extensions** is an application that differs from a **Conforming Application** only in that it uses non-standard facilities which are consistent with this standard. Such an application shall fully document its requirements for these extended facilities, in addition to the documentation required of a **Conforming Application.**

2.2.3 Language Conformance

As of this version of IEEE Std 1003.1, the standard has been described only in terms of the "C" programming language. In the future, it is expected that language bindings for other programming languages will be described as well.

2.2.3.1 C Language Binding

The *ANSI/X3.159-198x Programming Language C Standard* will be used as a basis for a C language binding to IEEE Std 1003.1. Included in the ANSI standard are definitions of C library functions that will be required upon its final adoption. Any C language implementation providing the facilities listed in chapter 8 of this standard shall be deemed conforming, provided that the implementation clearly states that its C language does not conform to *ANSI/X3.159-198x Programming Language C Standard* and its C implementation acts only as an interim binding.

The following rules apply to the usage of C language library functions; each of the statements in this section applies to the detailed function descriptions in Chapters 3 through 9, unless explicitly stated otherwise:

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

Any function declared in a header may also be implemented as a macro defined in the header, so a library function should not be declared explicitly if its header is included.

An application may use **`#undef`** to remove any macro's definition to insure that an actual function is referenced.

Any invocation of a library function that is implemented as a macro shall expand to code that evaluates each of its arguments only once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments.

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

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

2.3 General Terms

The following terms are used in this standard:

access mode
: An access mode is a form of access permitted to a file. Each implementation shall provide separate read, write, and execute/search access modes.

address space
: The range of memory locations that can be referenced by a **process**.

appropriate privileges
: Each implementation shall provide a means of associating privileges with a **process** with regard to the function calls and function call options defined in this standard that need special privileges.

background process
: A **process** that is not in the (non-zero) distinguished **process group** of its **controlling terminal**. See Job Access Control §7.1.1.5.

block special file
: A file that refers to a device. A **block special file** is normally distinguished from a **character special file** by providing a more structured interface to the device.

character special file
: A file that refers to a device. A **character special file** has no defined structure and its use is implementation defined.

child process

See **process**.

clock tick

A rate used within the system for scheduling and accounting. The rate is defined by {CLK_TCK}, which is the number of intervals per second.

controlling process

The **process group leader** that established the connection to the **controlling terminal**.

controlling terminal

A **terminal** that is associated with a **process group**. Certain input sequences from the **controlling terminal** (see **General Terminal Interface** §7.1) cause signals to be sent to all **processes** in the **process group** associated with the **controlling terminal**.

current working directory

See **working directory**.

device

A computer peripheral or an object that appears to the application as such.

directory

A **directory** is a **file** that contains **directory entries**. No two **directory entries** in the same **directory** shall have the same name.

directory entry (or link)

An object that associates a **filename** with a **file**. Several **directory entries** can associate names with the same **file**.

dot

The **filename** consisting of a single dot character (`.`). See **pathname resolution** §2.4.

dot-dot

The **filename** consisting solely of two dot characters (`..`). See **pathname resolution** §2.4.

effective group ID

An attribute of a **process** that is used in determining **file access permissions** (see **file access permissions** §2.4). See **group ID**. This value is subject to change during the **process lifetime**, as described in *setgid*() §4.2.2 and *exec* §3.1.2.

effective user ID

An attribute of a **process** that is used in determining **file access permissions** (see **file access permissions** §2.4). See **user ID**. This value is subject to change during the **process lifetime**, as described in *setuid*() §4.2.2 and *exec* §3.1.2.

Epoch

The **Epoch** refers to the time at 0 hours, 0 minutes, 0 seconds, Coordinated Universal Time on January 1, 1970. The value **seconds since the Epoch** refers to the difference in seconds between the referenced time and the **Epoch**, not counting leap seconds.

FIFO special file (or FIFO)

A **FIFO** special file is a file. Data written to a **FIFO** special file is read on a first-in-first-out basis. Other characteristics of **FIFOs** are described under *open*() §5.3.1, *read*() §6.4.1, *write*() §6.4.2, and *lseek*() §6.5.3.

file

An object that can be written to and/or read from. A file has certain attributes, including access permissions and type. File types include **regular file, character special file, block special file, FIFO special file,** and **directory.** Other types of files may be defined by the implementation.

file descriptor

A file descriptor is a per-process unique, non-negative integer used to identify a file for the purpose of file access.

file group class

A **process** is in the **file group class** of a file if the **process** is not in the **file owner class** and if the **effective group ID** or one of the **supplementary group IDs** of the **process** matches the **group ID** associated with the file. Other members may be implementation defined.

file mode

The **file mode** contains the **file permission bits** and other characteristics of the file, as described in <sys/stat.h> §5.6.1.

filename

Names consisting of 1 to {NAME_MAX} bytes may be used to name a file. The characters composing the name may be selected from the set of all character values excluding the slash character and those containing the null byte (octal zero). The filenames **dot** and **dot-dot** have special meaning; see **pathname resolution** §2.4. A **filename** is sometimes referred to as a **pathname component**.

file offset

The **file offset** specifies the position in the file where the next I/O operation begins. Each **open file description** associated with a **regular file** or **special file** has a **file offset**. There is no **file offset** specified for a **pipe** or **FIFO**.

file other class

A **process** is in the **file other class** if the **process** is not in the **file owner class** or **file group class**.

file owner class

A process is in the **file owner class** of a file if the effective user ID of the process matches the user ID of the file.

file permission bits

The **file permission bits** are used, along with other information, to determine if a process has read, write, or execute/search permission to a file. The bits are divided into three parts: owner, group, and other. Each part is used with the corresponding file class of processes. These bits are contained in the **file mode**, as described in <sys/stat.h> §5.6.1. The detailed usage of the **file permission** bits in access decisions is described in **file access permissions** §2.4.

file serial number

A **file serial number** is a per-file system unique identifier for a file. **File serial numbers** are not necessarily unique throughout the system.

file system

A collection of files and certain of their attributes. It provides a name space for file **serial numbers** referring to those files.

foreground process

A process that is in the (non-zero) distinguished **process group** of its **controlling terminal**. See Job Access Control §7.1.1.5.

group ID

Each system **user** is a member of at least one group. A group is identified by an integer known as a **group ID**, which must be between zero and {UID_MAX}, inclusive. When the identity of a group is associated with a **process**, a **group ID** value is referred to as a **real group** ID, an **effective group** ID, one of the (optional) **supplementary group** IDs, or an (optional) **saved set-group-ID**.

Job Control Option

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. **Conforming implementations** may optionally support **job control** facilities; the presence of this option is indicated to the application at compile time or run time by the definition of the {_POSIX_JOB_CONTROL} symbol; see **Symbolic Constants** §2.10). Portions of the standard operating system interface that are required only on implementations that support the **Job Control Option** are so labeled.

job control process group leader

A **job control process group leader** is a process that called the *jcsetpgrp*() function to become a **process group leader**. **Job control process group leaders** can exist on implementations that support the **Job Control Option**.

As contrasted with a session process group leader, a **job control process group leader** is one of a set of processes all belonging to the same process group that are typically controlled as a unit via the **Job Control Option** signaling mechanisms. While there is usually only one session process group leader per login session, there are usually many **job control process group leaders**. Side effects typically associated with login session creation and destruction that are performed for **session process group leaders** (such as effecting terminal affiliation) are not performed for **job control process group leaders**.

link

See directory entry.

link count

The **link count** of a file is the number of **directory entries** that refer to that **file**.

mode

The mode of a file is a collection of attributes that specifies the file's type and its access permissions. (See **file access permissions** §2.4).

open file

A file that is currently associated with a **file descriptor**.

open file description

An **open file description** records how a **process** or **group of processes** are accessing a file. Each file descriptor refers to exactly one **open file description**, but an **open file description** can be referred to by more than one file descriptor. A file offset, file status §6.5.1.2.5, and file access modes §6.5.1.2.6 are attributes of an **open file description**.

parent directory

A directory is known as a **parent directory** of all files that are referenced by its directory entries, with the exception of the **directory entries** for **dot** and **dot-dot**.

parent process

See **process**.

parent process ID

A new **process** is created by a currently active **process**. The **parent process ID** of a **process** is the **process ID** of its creator, for the lifetime of the creator. After the creator's lifetime has ended, the **parent process ID** is the **process ID** of an implementation defined **process**.

path prefix

A **path prefix** is a **pathname**, with an optional ending slash, that refers to a directory.

pathname

A **pathname** is a string that is used to identify a **file**. It consists of, at most, {PATH_MAX} bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more **filenames** separated by slashes. Multiple successive slashes are considered the same as one slash. The interpretation of the **pathname** is described under **pathname resolution** §2.4.

pathname component

See **filename**.

pipe

A **pipe** is an unnamed object created by the *pipe*,() *dup*,() or *fcntl*() functions that behaves identically to a **FIFO special file** for input and output.

portable filename character set

The following set of graphical characters shall be portable across **conforming implementations** of IEEE Std 1003.1:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -
```

The last three characters are the dot, underscore, and hyphen characters, respectively. The hyphen should not be used as the first character of a portable **filename**.

privilege

See **appropriate privileges**.

process

An **address** space and single thread of control that executes within that **address** space, and its required system resources. A **process** is created by another **process** issuing the *fork*() function. The **process** that issues *fork*() is known as the **parent process**, and the new **process** created by the *fork*() as the **child process**.

process ID

Each active **process** in the system is uniquely identified during its lifetime by a positive integer less than or equal to {PID_MAX} called a **process ID**. A **process ID** may be re-used by the system after the **process lifetime** ends, provided the process was not a **process group leader**. If a **process group leader's** lifetime ends, its **process ID** shall not be re-used until all **processes** in the **process group** terminate.

process group

Each active process is a member of a **process group** that is identified by a **process group ID**. A newly created **process** joins the **process group** of which its creator is a member.

process group ID

The **process group ID** is the **process ID** of the initial **process group leader**.

process group leader

A process group leader is a process whose process ID is the same as its process group ID. Any process that is not a process group leader may detach itself from its process group and become the process group leader of a new process group by calling either the *setpgrp*() or the *jcsetpgrp*() function, which can cause a process to become either a session process group leader or a job control process group leader, respectively. Job control process group leaders can exist on implementations that support the Job Control Option.

process lifetime

After a process is created with a *fork*() function, it is considered active. Its thread of control and address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the process ID are still in use. When another process executes a *wait*() or *wait2*() function for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends.

read-only file system

An implementation defined characteristic of a file system that restricts file system modifications.

real group ID

The attribute of a process that, at the time of process creation, identifies the group of the user who created the process. See group ID. This value is subject to change during the process lifetime, as described in *setgid*() §4.2.2.

real user ID

The attribute of a process that, at the time of process creation, identifies the user who created the process. See user ID. This value is subject to change during the process lifetime, as described in *setuid*() §4.2.2.

regular file

A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system.

root directory

A directory, associated with a process, that is used in pathname resolution §2.4 for pathnames that begin with a slash.

saved set-group-ID

When the saved set-group-ID option is implemented, the saved set-group-ID is an attribute of a process that allows some flexibility in the assignment of the effective group ID attribute, as described in *setgid*() §4.2.2, and *exec* §3.1.2.

saved set-user-ID

When the saved set-user-ID option is implemented, the saved set-user-ID is an attribute of a process that allows some flexibility in the assignment of the effective user ID attribute, as described in *setuid*() §4.2.2, and *exec* §3.1.2.

session process group leader

A session process group leader is a process that called the *setpgrp*() function to become a process group leader. When the Job Control Option is not implemented, this term is a synonym for process group leader. When the Job Control Option is implemented, this term is used to distinguish the functionality of the *setpgrp*() function from that of the *jcsetpgrp*() function, which establishes a job control process group leader.

As contrasted with a job control process group leader, there is typically only one session process group leader per login session and it is the main command interpreter for the session. All processes created during the session are descendants of the session process group leader and members of the same process group.

signal

A mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term signal is also used to refer to the event itself.

slash

The term slash is used to represent the literal character "/". This character is also known as "solidus" in ISO DIS 8895/1.

supplementary group ID

A process has up to {NGROUPS_MAX} supplementary group IDs used in determining file access permissions, in addition to the effective group ID. The supplementary group IDs of a process are set to the supplementary group IDs of the parent process when the process is created.

system

The term system is used in this standard to refer to an implementation of this standard.

system process

A process that runs on behalf of the system. It may have special implementation defined characteristics.

terminal (or terminal device)

A character special file that obeys the specifications of the General Terminal Interface §7.1.

terminal group ID

The attribute of a process that is used to identify the controlling terminal for a login session. All processes in a process group that have a controlling terminal share the same controlling terminal. That is, the terminal group ID is either cleared or has the same value for all processes in a process group.

user ID

Each system user is identified by an integer known as a user ID, which must be between zero and {UID_MAX}, inclusive. When the identity of a user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or an (optional) saved set-user-ID.

working directory (or current working directory)

A directory, associated with a process, that is used in pathname resolution §2.4 for pathnames that do not begin with a slash.

2.4 General Concepts

file access permissions

File access control is provided using the file permission bits along with other information. These bits are set at file creation, *open*() §5.3.1 or *creat*() §5.3.2, and are changed by *chmod*() §5.6.4. These bits are read by *stat*() or *fstat*() §5.6.2.

Whenever a process requests file access permission for read, write, or execute/search, the following applies:

> If the process has appropriate privileges to override the access mechanism:
>
> > If read, write, or directory search is requested, access is granted.
> >
> > If execute permission is requested, access is granted if at least one of the execute file permission bits is set, or if an implementation defined access mechanism is enabled that allows execute permission; otherwise, access is denied.
>
> Otherwise, the access mechanism is:
>
> > If the requested access permission bit is set in the part (owner/group/other) of the file permission bits that corresponds to the file class (owner/group/other) of the process, or if an implementation defined access mechanism is enabled that allows the requested permission, access is granted, unless the process is denied access by an implementation defined constraint.

Otherwise, access is denied.

An implementation may provide an alternative access mechanism, enabled explicitly by the user, that does not necessarily use the file permission bits. This alternative access mechanism shall:

- Specify appropriate file permission bits for the owner, group, and other classes of the file to be returned by *stat*() or *fstat*().
- Be enabled only by explicit user action.
- Be disabled after the file permission bits are changed by *chmod*().

file hierarchy

Files in the system are organized in a hierarchical structure in which all of the non-terminal nodes are directories and all of the terminal nodes are any other type of file. Because multiple directory entries may refer to the same file, the hierarchy is properly described as a *directed graph*.

filename portability

Filenames should be constructed from the portable filename character set because the use of other characters can be confusing or ambiguous in certain contexts.

file times update

Each file has three associated time values that are updated when file data has been accessed, file data has been modified, or file status has been changed, respectively. These values are returned in the file characteristics structure, as described in <sys/stat.h> §5.6.1.

For each function in this standard that reads or writes file data or changes the file status, the appropriate time-related fields are noted as "marked-for-update." At an update point in time, any marked fields are set to the current time and the update marks are cleared. One such update point is when the file is no longer open by any process. Additional update points are implementation defined. Updates are not done for files on read-only file systems.

pathname resolution

Pathname resolution is performed for a process to resolve a pathname to a particular file in a file hierarchy. There may be multiple pathnames that resolve to the same file.

Each filename in the pathname is located in the directory specified by its predecessor (for example, in the pathname fragment "a/b", file "b" is located in directory "a"). Pathname resolution fails if this cannot be accomplished. If the pathname begins with a slash, the predecessor of the first **filename** in the **pathname** is taken to be the **root directory** of the **process** (such **pathnames** are referred to as **absolute pathnames**). If the **pathname** does not begin with a **slash**, the predecessor of the first **filename** of the **pathname** is taken to be the **current working directory** of the **process** (such **pathnames** are referred to as **relative pathnames**).

The interpretation of a **pathname component** is dependent on the values of {NAME_MAX} and {_POSIX_NO_TRUNC} associated with the **path prefix** of that component. If any **pathname component** is longer than {NAME_MAX}, and {_POSIX_NO_TRUNC} is in effect for the **path prefix** of that component (see *pathconf*() §5.7.1), the implementation shall consider this an error condition. Otherwise, the implementation shall use the first {NAME_MAX} bytes of the **pathname component**.

The special **filename**, **dot**, refers to the **directory** specified by its predecessor. The special **filename**, **dot-dot**, refers to the **parent directory** of its predecessor **directory**. As a special case, in the **root directory**, **dot-dot** may refer to the **root directory** itself.

A **pathname** consisting of a single **slash** resolves to the **root directory** of the **process**. If {_POSIX_PATHNAME_NULL} is defined, a null **pathname** (a **pathname** consisting of a null string) resolves to the **current working directory** of the process; otherwise, a null **pathname** is invalid.

2.5 Error Numbers

Most functions provide an error number in the external variable *errno*, which is defined as:

```
extern int errno;
```

This variable is defined only after calls to functions for which it is explicitly stated to be set. The variable *errno* should only be examined when it is indicated to be valid by a function's return value. No function defined in this standard sets *errno* to zero to indicate an error.

If more than one error occurs in processing a function call, this standard does not define in what order the errors are detected; therefore, any one of the possible errors may be returned.

Implementations may support additional errors not included in this list, may generate errors included in this list under circumstances other than those described here, or may contain extensions or limitations that prevent some errors from occurring. The **Errors** subsection in each function description specifies which error conditions shall be required and which may be implementation defined. Implementations shall not generate an error number different from the ones described here for error conditions described in this standard.

The following symbolic names identify the possible error numbers, in the context of functions specifically defined in this standard; these general descriptions are more precisely defined in the **Errors** sections of functions that return them. Only these symbolic names should be used in programs, since the actual value of the error number is implementation defined. All values shall be unique numbers. The implementation defined values for these names can be found in the header **<errno.h>**.

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

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

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

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

[EBUSY] Resource busy
An attempt was made to make use of a system resource that is not currently available because it is being used by another process in a manner that would conflict with the request being made by this process.

[ECHILD] No child processes
A *wait*() or *wait2*() function was executed by a process that had no existing or unwaited-for child processes.

[EDEADLK] Resource deadlock would occur
A process that has locked a system resource would have been put to sleep while attempting to access a resource locked by another process.

[EDOM] Domain error
Defined in *ANSI/X3.159-198x Programming Language C Standard*; an input argument is outside the defined domain of the

mathematical function.

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

[EFAULT] Bad address
The system detected an invalid address in attempting to use an argument of a call. The reliable detection of this error is implementation defined; however, implementations that do detect this condition shall use this value.

[EFBIG] File too large
The size of a file would exceed an implementation defined maximum file size.

[EINTR] Interrupted function call
An asynchronous signal (such as SIGINT or SIGQUIT; see the description of header <signal.h> §3.3.1) was caught by the process during the execution of an interruptible function. If the signal handler performs a normal return, the interrupted function call may return this error condition.

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

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

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

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

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

[ENAMETOOLONG] Filename too long
The size of a pathname string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while

{_POSIX_NO_TRUNC} is in effect.

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

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

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

[ENOEXEC] Exec format error
A request is made to execute a file that, although it has the appropriate permissions, is not in the proper format.

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

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

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

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

[ENOTEMPTY] Directory not empty
A directory with entries other than dot and dot-dot was supplied when an empty directory was expected.

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

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

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

[EPIPE] Broken pipe
A write on a pipe or FIFO for which there is no process to read the data. This condition normally generates the signal SIGPIPE; the error is returned if the signal is ignored.

[ERANGE] Result too large
Defined in *ANSI/X3.159-198x Programming Language C Standard*; the result of the function is too large to fit in the available space.

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

[ESPIPE] Invalid seek
An *lseek*() function was issued on a pipe or FIFO.

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

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

2.6 Primitive System Data Types

Some data types used by the various system functions are not defined as part of this standard, but are defined by the implementation. These types are then defined in the header **<sys/types.h>**, which contains definitions for at least the following types:

Defined Type	Description	
clock_t	Used for system times (in {CLK_TCK}ths of a second)	8
dev_t	Used for device numbers	
ino_t	Used for file serial numbers	
mode_t	Used for some file attributes, e.g. file type, file access permissions	9
nlink_t	Used for link counts	9
off_t	Used for file sizes	
time_t	Used for system times (in seconds)	
uid_t	Used for user IDs and group IDs	9

All of the types listed above shall be integral types. B

Additional type definitions may also be given in this header. C

2.7 Environment Description

An array of strings called the environment is made available when a process begins. This array is pointed to by the external variable *environ*, which is defined as:

```
extern char **environ;
```

These strings have the form ''*name=value*''. There is no meaning associated with the order of the strings in the environment. If more than one string in a process's environment has the same *name*, the consequences are undefined. The following names may be defined and have the indicated meaning if they are defined: C A

HOME	Name of the user's initial working directory, from the password database (see description of the header **<pwd.h>** §9.2.2).	
IFS	Characters used as field separators. The format of this string is currently not defined as part of this standard.	A
LANG	Specifies the name of the pre-defined setting for locale.	C
LC_CTYPE	Specifies the name of the locale for character classification.	C
LC_COLLATE	Specifies the name of the locale for collation information.	C

LC_TIME	Specifies the name of the locale for date/time formatting information.
LC_NUMERIC	Specifies the name of the locale containing numeric editing (i.e., radix character) information.
LOGNAME	The name of the user's login account, corresponding to the login name in the password database (see description of the header **<pwd.h>**).
MAIL	System mailer information. The format of this string is currently not defined as part of this standard.
PATH	The sequence of path prefixes that certain commands and functions apply in searching for a file known by an incomplete pathname (a pathname without a leading slash). The prefixes are separated by a colon (**:**). When a non-zero-length prefix is applied to an incomplete pathname, a slash is inserted between the prefix and the incomplete pathname. A zero-length prefix is a special prefix that indicates the current working directory. It appears as two adjacent colons ("**::**"), as an initial colon preceding the rest of the list, or as a trailing colon following the rest of the list. The list is searched from left to right until an executable program by the specified name is found. If the filename being sought contains a slash, the search through path prefixes is not done.
PS1	Prompting string for interactive programs. The format of this string is currently not defined as part of this standard.
PS2	Prompting string for interactive programs. The format of this string is currently not defined as part of this standard.
SHELL	The shell command interpreter name. The format of this string is currently not defined as part of this standard.
TERM	The terminal type for which output is to be prepared. This information is used by commands and application programs wishing to exploit special capabilities specific to a terminal.
TZ	Time zone information. The format of this string is defined in *asctime*() §8.1.1.

It is recommended that the environment variable *name*s consist solely of characters from the portable filename character set. Other valid characters may be permitted by an implementation, but use of them by an application may limit its portability. Upper- and lowercase letters retain their unique identities and are not folded together. It is recommended that only capital letters, underscores, and numbers be used for environment variable names and that the first character be a letter.

The *values* that the environment variables may be assigned are not restricted except that they are considered to end with a null byte and the total space used to store the environment and the arguments to the process is limited to {ARG_MAX} bytes.

Other *name=value* pairs may be placed in the environment by manipulating the *environ* variable or by using *envp* arguments when creating a process (see *exec* §3.1.2).

2.8 C Language Definitions

Certain terms used in this standard are considered to be defined by the C programming language. The following terms are defined in the *ANSI/X3.159-198x Programming Language C Standard* (see **C Language Standard** §A.2.1):

- NULL
- **byte**
- **character**
- **character array**
- **string**
- **empty string**

The term NULL **pointer** in this standard is equivalent to the term **null pointer** used in the *ANSI/X3.159-198x Programming Language C Standard.*

2.9 Numerical Limits

The following subsections list magnitude limitations imposed by a specific implementation. A standard conforming implementation shall define each of the values specified below as a symbolic constant in the header **<limits.h>**. The values given below shall be replaced by restricted constant expressions suitable for use in `#if` preprocessing directives. The braces notation, {LIMIT}, is used in the standard to indicate these values, but the braces are not part of the name.

2.9.1 C Language Limits

Certain limits used in this standard are considered to be defined in the C programming language. The following limits are defined in the *ANSI/X3.159-198x Programming Language C Standard* (see **C Language Standard** §A.2.1):

CHAR_BIT
CHAR_MAX
CHAR_MIN
CLK_TCK
INT_MAX
INT_MIN
LONG_MAX
LONG_MIN
SCHAR_MAX
SCHAR_MIN
SHRT_MAX
SHRT_MIN
UCHAR_MAX
UINT_MAX
ULONG_MAX
USHRT_MAX

2.9.2 Run-Time Invariant Values

The following magnitude limitations shall be fixed for a specific implementation. A **Strictly Conforming Application** shall assume that the value supplied by **<limits.h>** in a specific implementation is that which pertains whenever the **Strictly Conforming Application** is run under that implementation. A specific instance of a specific implementation shall not vary the value from that supplied by **<limits.h>** for that implementation.

Name	Description	Minimum Value
MAX_INPUT	Maximum number of bytes allowed in a terminal input queue	256
NGROUPS_MAX	Maximum number of simultaneous supplementary group IDs per process	0
PASS_MAX	Maximum number of bytes in a password (not a string length; does not include a terminating null)	8
PID_MAX	Maximum value for a process ID	30000
UID_MAX	Maximum value for a user or group ID	32000

2.9.3 Run-Time Invariant Values (Possibly Indeterminate)

A definition of one of the following values shall be omitted from the **<limits.h>** on specific implementations where the corresponding value is equal to or greater than the stated minimum, but is indeterminate. This depends, for example, on the amount of available memory space on a specific instance of a specific implementation.

Name	Description	Minimum Value
ARG_MAX	Maximum length of arguments for *exec*() in bytes, including *environ* data	4096
CHILD_MAX	Maximum number of simultaneous processes per user ID	6
MAX_CANON	Maximum number of bytes in a terminal canonical input line. (See Canonical Mode Input Processing §7.1.1.7.)	256
OPEN_MAX	Maximum number of files that one process can have open at any given time	16

2.9.4 Pathname Variable Values

The following values may be constants within an implementation, or may vary from one pathname to another. For example, file systems or directories may have different characteristics.

A definition of one of the following values shall be omitted from the **<limits.h>** on specific implementations where the corresponding value is equal to or greater than the stated minimum, but is indeterminate. The actual value supported for a specific pathname shall be provided by the *pathconf*() §5.7.1 function.

Name	Description	Minimum Value
NAME_MAX	Maximum number of bytes in a file name (not a string length; does not include a terminating null).	14
PATH_MAX	Maximum number of bytes in a pathname (not a string length; does not include a terminating null).	255

2.9.5 Run-Time Increasable Values

The following magnitude limitations shall be fixed by specific implementations. A **Strictly Conforming Application** shall assume that the value supplied by **<limits.h>** in a specific implementation is the minimum that pertains whenever the **Strictly Conforming Application** is run under that implementation. A specific instance of a specific implementation may increase the value relative to that supplied by **<limits.h>** for that implementation.

Name	Description	Minimum Value	
LINK_MAX	Maximum value of a file's link count	8	c c
PIPE_BUF	Maximum number of bytes that is guaranteed to be atomic when writing to a pipe	512	c c c

c

2.10 Symbolic Constants

A conforming implementation shall have a header with the name **<unistd.h>**. This file defines the symbolic constants and structures referenced elsewhere in the standard and not already defined or declared in some other header. When used, it shall be referenced as follows:

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

The constants defined in this file are shown below. The actual values of the constants are implementation defined.

2.10.1 Symbolic constants for the *access*() function

Constant	Description
R_OK	Test for read Permission
W_OK	Test for write Permission
X_OK	Test for execute or search Permission
F_OK	Test for existence of file

2.10.2 Symbolic constant for the *lseek*() function

Constant	Description
SEEK_SET	Set file offset to *offset*
SEEK_CUR	Set file offset to current plus *offset*
SEEK_END	Set file offset to EOF plus *offset*

2.10.3 Compile time symbolic constants for portability specifications

These constants may be used by the application, at compile time, to determine which optional facilities are present and what actions shall be taken by the implementation.

Some of these symbols may have more liberal, or less restrictive, values at the time of execution. Although a Strictly Conforming Application can rely on the values compiled from the <unistd.h> header to afford it portability on all instances of an implementation, it may choose to interrogate a value at run time to take advantage of the current configuration. See *sysconf*() §4.8.1.

{_POSIX_EXIT_SIGHUP}
: If defined, if the process is a session process group leader, the *_exit*() §3.2.2 function will send the SIGHUP signal to all processes with group IDs equal to that of the calling process.

{_POSIX_JOB_CONTROL}
: If this symbol is defined, it indicates that the implementation supports the Job Control Option.

{_POSIX_KILL_PID_NEG1}
: If defined, a *kill*() §3.3.2 function call with *pid* of −1 will send the signal to the sending process; otherwise, the sending process will be excluded.

{_POSIX_KILL_SAVED}
: If defined, and if {_POSIX_SAVED_IDS} is also defined, the *kill*() §3.3.2 uses the saved set-user-ID instead of the effective user-ID.

{_POSIX_PATHNAME_NULL}
: If defined, a null pathname resolves to the current working directory; otherwise, a null pathname is considered invalid.

{_POSIX_PGID_CLEAR}
: If defined, if the process is a session process group leader, the

_exit() §3.2.2 function will cause all process group IDs equal to that of the calling process to have their process group IDs set to zero.

{_POSIX_SAVED_IDS}
If defined, the *exec*() §3.1.2 saves the effective user and group IDs.

{_POSIX_VERSION}
The integer value 198803. This value will change with each published version or revision of this standard to indicate the (4-digit) year and (2-digit) month that the standard was approved by the IEEE Standards Board. *Editor's Note: The value 198803 is tentative as of this draft. The published Full Use Standard will contain the value that should be used by applications; however, it is guaranteed to not be less than 198803.*

2.10.4 Execution time symbolic constants for portability specifications

These constants may be used by the application, at execution time, to determine which optional facilities are present and what actions shall be taken by the implementation in some circumstances described by this standard as **implementation defined**.

If any of the following constants are not defined in the header **<unistd.h>**, the value varies depending on the file to which it is applied. See *pathconf*() §5.7.1.

If any of the following are defined to have value −1 in the header **<unistd.h>**, the implementation shall not provide the option on any file. If any of the following are defined to have a value other than −1 in the header **<unistd.h>**, the implementation shall provide the option on all applicable files.

All of the following, whether defined in **<unistd.h>** or not, may be queried with respect to a specific file using the *pathconf*() or *fpathconf*() functions.

{_POSIX_CHOWN_RESTRICTED}
The use of the *chown*() §5.6.5 function is restricted to a process with appropriate privileges.

{_POSIX_CHOWN_SUP_GRP}
The use of the *chown*() §5.6.5 function is restricted to changing the group ID of a file only to the effective group ID of the process or to one of its supplementary group IDs.

{_POSIX_DIR_DOTS}
An "empty directory" contains entries for dot and dot-dot; otherwise it must be completely empty.

{_POSIX_GROUP_PARENT}
A newly created file, directory, or FIFO receives the group ID of its parent directory; otherwise, the process's effective group ID is

used.

{_POSIX_LINK_DIR}
: Any user is allowed to *link*() §5.3.4 or *unlink*() §5.5.1 directories.

{_POSIX_NO_TRUNC}
: Pathname components longer than {NAME_MAX} generate an error.

{_POSIX_UTIME_OWNER}
: The owner of a file is allowed to use the *utime*() §5.6.6 function with a non-NULL argument.

{_POSIX_V_DISABLE}
: Terminal special characters defined in <termios.h> §7.1.2 can be disabled using this character value, if it is defined. See *tcgetattr*() and *tcsetattr*() §7.2.1.

3. Process Primitives

The functions described in this chapter perform the most primitive operating system services dealing with processes, interprocess signals, and timers. All attributes of a
process that are specified in this standard shall remain unchanged by a process primitive 8
unless the description of that process primitive states explicitly that the attribute is changed.

3.1 Process Creation

Running a program takes two steps: first, the *fork*() function is called to produce a new process, then that new process calls one of the *exec* functions to start the new program.

3.1.1 Process Creation

Function: *fork*()

3.1.1.1 Synopsis

```
int fork ( )
```

3.1.1.2 Description

The *fork*() function shall cause creation of a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except for the following:

The child process has a unique process ID. If the implementation supports the Job B
Control Option, the child process ID also does not match any active process group B
ID. B

The child process has a different parent process ID (which is the process ID of the parent process).

The child process has its own copy of the parent's file descriptors. Each of the C
child's file descriptors refers to the same open file description with the C
corresponding file descriptor of the parent. C

The child process's values of *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* are set to zero (see *times*() §4.5.2).

File locks previously set by the parent are not inherited by the child. (See *fcntl*() §6.5.2.)

Pending alarms are cleared for the child process. (See *alarm*() §3.4.1.)

The set of signals pending for the child process is initialized to the empty set. (See <signal.h> §3.3.1.)

All other process characteristics defined by this standard shall be the same in the parent and the child processes. The inheritance of process characteristics not defined by this standard is implementation defined and shall be documented in the system documentation. (See Documentation §2.2.1.2.)

If during the *fork*() function call, a signal is directed to a group of processes of which the child process is a member, whether or not the signal is delivered to the child process is undefined. (See *kill*() §3.3.2.)

3.1.1.3 Returns

Upon successful completion, *fork*() shall return to the child process a value of zero and shall return to the parent process the process ID of the child process, and both processes shall continue to execute from the *fork*() function. Otherwise, a value of −1 shall be returned to the parent process, no child process shall be created, and *errno* shall be set to indicate the error.

3.1.1.4 Errors

If any of the following conditions occur, the *fork*() function shall return −1 and set *errno* to the corresponding value:

[EAGAIN] The system lacked the necessary resources to create another process, or; the system-imposed limit on the total number of processes under execution by a single user would be exceeded.

For each of the following conditions, if the condition is detected, the *fork*() function shall return −1 and set *errno* to the corresponding value:

[ENOMEM] The process requires more space than the system is able to supply.

3.1.1.5 References

alarm() §3.4.1, *exec* §3.1.2, *fcntl*() §6.5.2, *kill*() §3.3.2, *times*() §4.5.2, *wait* §3.2.1.

3.1.2 Execute a File

Functions: *execl*(), *execv*(), *execle*(), *execve*(), *execlp*(), *execvp*()

3.1.2.1 Synopsis

```
int execl (path, arg0, arg1, ..., argn, (char *) 0)
char *path, *arg0, *arg1, ..., *argn;

int execv (path, argv)
char *path, *argv[ ];

int execle (path, arg0, arg1, ..., argn, (char *) 0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[ ];

int execve (path, argv, envp);
char *path, *argv[ ], *envp[ ];

int execlp (file, arg0, arg1, ..., argn, (char *) 0)
char *file, *arg0, *arg1, ..., *argn;

int execvp (file, argv)
char *file, *argv[ ];

extern char **environ;
```

3.1.2.2 Description

The *exec* family of functions shall replace the current process image with a new process image. The new image is constructed from a regular, executable file called the *new process image file*. There shall be no return from a successful *exec*, because the calling B process image is overlaid by the new process image.

When a C program is executed as a result of this call, it shall be entered as a C language procedure call as follows:

```
extern char **environ;
int main (argc, argv)                                   B
int argc;
char **argv;
```

where *argc* is the argument count (one or greater), *argv* is an array of character pointers to the arguments themselves and *environ* is an array of character pointers to the B environment strings. The *environ* array is terminated by a NULL pointer. C

The arguments specified by a program with one of the *exec* functions shall be passed on to the new process image in the corresponding *main*() arguments.

The argument *path* points to a pathname that identifies the new process image file.

The argument *file* points to the new process image file. If the *file* argument does not B contain a slash character, the path prefix for this file is obtained by a search of the directories passed as the environment variable PATH (see Environment Description

§2.7). If this environment variable is not present, the results of the search are implementation defined.

The arguments *arg0*, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process image. The list is terminated by a NULL pointer. The argument *arg0* should point to a filename that is associated with the process being started by one of the *exec* functions.

The argument *argv* is an array of character pointers to null-terminated strings. The last member of this array shall be a NULL pointer. These strings constitute the argument list available to the new process image. The value in *argv*[0] should point to a filename that is associated with the process being started by one of the *exec* functions.

The argument *envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The *envp* array is terminated by a NULL pointer.

For those forms not containing an *envp* pointer (*execl()*, *execv*(), *execlp*(), and *execvp*()) the environment is taken from the external variable *environ*.

The number of bytes available for the new process's combined argument and environment lists is {ARG_MAX}. The implementation shall specify in the system documentation (see Documentation §2.2.1.2) whether null terminators, pointers, and/or any alignment bytes, are included in this total.

File descriptors open in the calling process image remain open in the new process image, except for those whose close-on-exec flag FD_CLOEXEC is set (see *fcntl*() §6.5.2, <fcntl.h> §6.5.1). For those file descriptors that remain open, all attributes of the open file description remain unchanged.

The file locks held by a process are not affected by the *exec* functions. See *fcntl*() §6.5.2.

Signals set to the default action (SIG_DFL) in the calling process image shall be set to the default action in the new process image. Signals set to be ignored (SIG_IGN) by the calling process image shall be set to be ignored by the new process image. Signals set to be caught by the calling process image shall be set to the default action in the new process image (see *sigaction*() §3.3.4).

If the set-user-ID mode bit of the new process image file is set (see *chmod*() §5.6.4), the effective user ID of the new process image is set to the owner ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. If {_POSIX_SAVED_IDS} is defined, the effective user ID and effective group ID of the new process shall be saved (as the *saved set-user-ID* and the *saved set-group-ID*) for use by the *setuid*() function.

The new process image also inherits at least the following attributes from the calling process image:

- process ID
- parent process ID
- process group ID
- terminal group ID
- time left until an alarm clock signal (see *alarm*() §3.4.1)
- current working directory
- root directory
- file mode creation mask (see *umask*() §5.3.3)
- process signal mask (see *sigprocmask*() §3.3.5)
- pending signals (see *sigpending*() §3.3.6)
- *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see *times*() §4.5.2)

Upon successful completion, the *exec* functions shall mark for update the *st_atime* field of the file. If the *exec*() function failed but was able to locate the *process image file*, whether the *st_atime* field is marked for update is unspecified.

3.1.2.3 Returns

If one of the *exec* functions returns to the calling process image, an error has occurred; the return value shall be −1, and *errno* shall be set to indicate the error.

3.1.2.4 Errors

If any of the following conditions occur, the *exec* functions shall return −1 and set *errno* to the corresponding value:

[E2BIG] The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit of {ARG_MAX} bytes.

[EACCES] Search permission is denied for a directory listed in the new process image file's path prefix, or the new process file is not a regular file, or the new process image file denies execution permission.

[ENAMETOOLONG]
The length of the *path* or *file* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOENT] One or more components of the new process image file's pathname do not exist.

[ENOEXEC] The new process image file has the appropriate access permission, but is not in the proper format.

[ENOTDIR] A component of the new process image file's path prefix is not a directory.

For each of the following conditions, if the condition is detected, the functions shall return −1 and return the corresponding value in *errno*:

[ENOMEM] The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.

3.1.2.5 References

alarm() §3.4.1, *chmod*() §5.6.4, *_exit*() §3.2.2, *fcntl*() §6.5.2, *fork*() §3.1.1, <signal.h> §3.3.1, *sigprocmask*() §3.3.5, *sigpending*() §3.3.6, *stat*() §5.6.2, <sys/stat.h> §5.6.1, *times*() §4.5.2, *umask*() §5.3.3, Environment Description §2.7.

3.2 Process Termination

There are three kinds of process termination:

Normal termination occurs by a return from *main*() or when requested with the *exit*() or *_exit*() functions.

Simple abnormal termination occurs when some signals are received (see <signal.h> §3.3.1).

Abnormal termination with actions occurs when requested with the *abort*() function or when other signals are received. Actions taken, if any, are implementation defined.

The *exit*() and *abort*() functions shall be as described in the *ANSI/X3.159-198x Programming Language C Standard* (see C Language Standard §A.2.1). Both *exit*() and *abort*() shall terminate a process with the consequences specified in *_exit*() §3.2.2, except that the status made available to *wait*() or *wait2*() by *abort*() shall be that of a process terminated by the SIGABRT signal.

A parent process can suspend its execution to wait for termination of a child process with the *wait*() or *wait2*() functions.

3.2.1 Wait for Process Termination

Functions: *wait*(), *wait2*() 8

3.2.1.1 Synopsis

```
int wait (stat_loc)
int *stat_loc;
                                                    B
#include <sys/wait.h>                               8
int wait2 (stat_loc, options)                       8
int *stat_loc;                                      8
int options;                                        8
```

3.2.1.2 Description

The header <sys/wait.h> defines the following arguments for the *wait2*() function: 8

Constant	Description (*wait2*() only)	
WNOHANG	return immediately if no children to wait for	8
WUNTRACED	also return status for stopped children	8

The *wait*() function suspends execution of a process until one of its children terminates. The termination of a child process causes *wait*() to return. If several child processes c have terminated, which child's information is returned by a call to *wait*() is unspecified. c Signals or implementation defined conditions may cause the return of *wait*() prior to the termination of a child. If a child process has terminated prior to the call on *wait*(), return shall be immediate.

If *stat_loc* is not (int *) 0, information called *status* shall be stored in the location pointed to by *stat_loc* as follows:

> If the child process terminated due to an *_exit*() function, the low order 8 bits of *status* (corresponding to the octal value 0377) shall be zero, and the 8 bits corresponding to the octal value 0177400 shall contain the low order 8 bits of the argument that the child process passed to *_exit*() (see *_exit*() §3.2.2).

> If the child process terminated due to a signal that was not caught, the low order 6 bits of *status* (corresponding to the octal value 077) shall contain the number of the signal that caused the termination, and the 8 bits corresponding to the octal value 0177400 shall be zero. In addition, if the bit that would be masked by the octal value 0200 is set, an abnormal termination with actions occurred (see c <signal.h> §3.3.1). c

> If the *wait*() function returned due to an implementation defined condition, the bit of *status* corresponding to the octal value 0100 shall be set. The value of the other bits of *status* are implementation defined and the child may not have terminated. If the child has terminated, a subsequent *wait*() function shall return

its *status*.

If a parent process terminates without waiting for its child processes to terminate, its children shall be assigned a new parent process ID corresponding to an implementation defined system process. The *wait*() function shall only return successfully on the termination of a child process or due to an implementation defined change in status of a child process.

If the implementation supports the Job Control Option, the *wait2*() function shall be provided as an alternate interface to provide both non-blocking status collection and the collection of the status of children that are stopped. The *stat_loc* argument is defined as above. If the *options* argument is zero, the behavior shall be identical to *wait*(). Otherwise, the *options* argument consists of the logical OR of the following flags:

WNOHANG Return immediately, even if there are no children to wait for. In this case, a return value of zero shall indicate that no children have terminated (or stopped, if WUNTRACED is also set).

WUNTRACED Return the status of stopped children. If the child process has stopped due to the delivery of a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal, its status may be collected using this option.

If WUNTRACED is set and the *status* of a stopped child process is reported, the 8 bits of *status* (corresponding to the octal value 0177400) shall contain the number of the signal that caused the process to stop and the low order 8 bits (corresponding to the octal value 0377) shall be set to the octal value 0177.

3.2.1.3 Returns

If the *wait*() function returns due to the receipt of a signal by the calling process, a value of −1 shall be returned to the calling process and *errno* shall be set to [EINTR]. If the *wait*() function returns due to a terminated child process, the process ID of the child shall be returned to the calling process. Otherwise, a value of −1 shall be returned, and *errno* shall be set to indicate the error.

If *wait2*() is called, the WNOHANG option is used, and there are no stopped or terminated children, then a value of zero is returned. Otherwise, a value of −1 is returned and *errno* shall be set to indicate the error.

3.2.1.4 Errors

If any of the following conditions occur, the *wait*() and *wait2*() functions shall return −1 and set *errno* to the corresponding value:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EINTR] The *wait*() function was terminated by a signal. The value pointed to by *stat_loc* may be undefined.

If any of the following conditions occur, the *wait2*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The *wait2*() was called with an invalid *options* value.

3.2.1.5 References

exec §3.1.2, *_exit*() §3.2.2, *fork*() §3.1.1, *pause*() §3.4.2, *times*() §4.5.2, *sigaction*() §3.3.4.

3.2.2 Terminate a Process

Function: *_exit*()

3.2.2.1 Synopsis

```
void _exit (status)
int status;
```

3.2.2.2 Description

The *_exit*() function shall terminate the calling process with the following consequences:

All open file descriptors in the calling process are closed.

If the parent process of the calling process is executing a *wait*() or *wait2*(), it is notified of the calling process's termination and the low order 8 bits of *status* are made available to it; see *wait* §3.2.1.

If the parent process of the calling process is not executing a *wait*() or *wait2*() function, the exit *status* code is saved for return to the parent process whenever the parent process executes a subsequent *wait*() or *wait2*().

Termination of a process does not terminate its children. Children of a terminated process shall be assigned a new parent process ID, corresponding to an implementation defined system process.

If the implementation supports the SIGCLD signal, a SIGCLD shall be sent to the parent process.

If the process is a controlling process, and if {_POSIX_EXIT_SIGHUP} is defined, the SIGHUP signal shall be sent to each process that has a process group ID equal to that of the calling process; otherwise, the signal shall not be sent.

If the process is a session process group leader, and if {_POSIX_PGID_CLEAR} is defined, the the process group ID shall be set to zero for each process that had a process group ID equal to that of the calling process; otherwise, the group IDs shall not be affected.

If the implementation supports the Job Control Option and if the calling process has child processes that are stopped, they shall be sent SIGHUP and SIGCONT signals.

If the implementation supports the Job Control Option, and if the process is a controlling process, the terminal group ID shall be cleared of all processes that match the terminal group ID of the calling process.

These consequences shall occur on process termination for any reason.

Application programs should use the C language function *exit*(), defined in the *ANSI/X3.159-198x Programming Language C Standard*, rather than *_exit*(). The function *_exit*() is included to clearly define the termination consequences for all processes. If a program reaches the end of a *main*() procedure, the return value is undefined.

3.2.2.3 Returns

The *_exit*() function cannot return to its caller.

3.2.2.4 References

close() §6.3.1, *sigaction*() §3.3.4, *wait* §3.2.1.

3.3 Signals

3.3.1 Signal Names

3.3.1.1 Synopsis

```
#include <signal.h>
```

3.3.1.2 Description

The <signal.h> header declares the *sigset_t* type and the *sigaction* structure. It also defines the following symbolic constants, each of which expands to a distinct constant expression of the type *void(*)()*, whose value matches no declarable function.

Symbolic Constant	Description
SIG_DFL	request for default signal handling
SIG_IGN	request that signal be ignored

The type *sigset_t* is used to represent sets of signals. It is always an integral or structure type. Several functions used to manipulate objects of type *sigset_t* are defined in *sigsetops* §3.3.3.

The <signal.h> header also declares the constants that are used to refer to the signals that occur in the system. Each of the signals defined by this standard shall have distinct, positive integral values. The value zero is reserved for use as the null signal (see *kill*() §3.3.2). An implementation may define additional signals that may occur in the system.

The following constants shall be defined by all implementations:

Required Signals

Symbolic Constant	Default Action	Description
SIGABRT	2	abnormal termination signal, such as is initiated by the *abort*() function (as defined in the *ANSI/X3.159-198x Programming Language C Standard*)
SIGALRM	1	timeout signal, such as initiated by the *alarm*() function (see *alarm*() §3.4.1)
SIGFPE	2	erroneous arithmetic operation, such as division by zero or an operation resulting in overflow
SIGHUP	1	hangup detected on controlling terminal (see Modem Disconnect §7.1.1.11) or death of process group leader (see *_exit*() §3.2.2)
SIGILL	2	detection of an invalid hardware instruction
SIGINT	1	interactive attention signal (see Special Characters §7.1.1.10)
SIGKILL	1	termination signal (cannot be caught or ignored)
SIGPIPE	1	write on a pipe with no readers (see *write*() §6.4.2)
SIGQUIT	2	interactive termination signal (see Special Characters §7.1.1.10)
SIGSEGV	2	detection of an invalid memory reference
SIGTERM	1	termination signal
SIGUSR1	1	reserved as application defined signal 1
SIGUSR2	1	reserved as application defined signal 2

In addition, if the implementation supports the Job Control Option, the following constants shall be defined:

Job Control Option Signals

Symbolic Constant	Default Action	Description
SIGCLD	3	child process terminated (see *_exit*() §3.2.2)
SIGCONT	5	continue if stopped (cannot be ignored)
SIGSTOP	4	stop signal (cannot be caught or ignored)
SIGTSTP	4	interactive stop signal (see **Special Characters** §7.1.1.10)
SIGTTIN	4	background read attempted from control terminal (see **Job Access Control** §7.1.1.5)
SIGTTOU	4	background write attempted to control terminal (see **Job Access Control** §7.1.1.5)

The constant SIGCLD may be defined in implementations that do not not support the Job Control Option. If SIGCLD is defined, it shall behave as specified in this standard.

Default actions for the preceding tables are as follows:

1 Simple abnormal termination (see **Process Termination** §3.2).

2 Abnormal termination with actions (see **Process Termination** §3.2).

3 Ignore the signal.

4 Stop the process if it is currently executing; otherwise, ignore the signal.

5 Continue the process if it is currently stopped; otherwise, ignore the signal.

A signal is said to be generated for (or sent to) a process when the event that causes the signal first occurs. Examples of such events include detection of hardware faults, timer expiration, and terminal activity; as well as the invocation of the *kill*() function. The same event may generate signals for multiple processes.

Each process has an action to be taken in response to each signal defined by the system. A signal is said to be delivered to a process when the appropriate action for the process and signal is taken. The action taken in response to a signal is determined at the time the signal is delivered. This determination is independent of the means by which the signal was originally generated.

During the time between the generation of a signal and its delivery, the signal is said to be pending. Ordinarily, this interval cannot be detected by an application. However, a signal can be blocked from delivery to a process, in which case it remains pending until it is unblocked. Each process has a signal mask that defines the set of signals currently blocked from delivery to it. The signal mask for a process is initialized from that of its parent. The *sigaction*(), *sigprocmask*(), and *sigsuspend*() functions control the manipulation of the signal mask. If a subsequent occurrence of a pending signal is generated, it is implementation defined as to whether the signal is delivered more than once.

When SIGCONT is generated for a process, all pending stop signals (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) for that process shall be discarded. Conversely, when any stop signal is generated for a process, any pending SIGCONT signals for that process shall be discarded.

An implementation shall document any conditions not specified by this standard under which the implementation generates signals. (See Documentation §2.2.1.2.)

3.3.1.3 Signal Actions

There are three types of actions that can be associated with a signal: SIG_DFL, SIG_IGN, or a *pointer to a function*. Initially, all signals shall be set to SIG_DFL or SIG_IGN prior to entry of the *main*() routine (see *exec* §3.1.2). The actions prescribed by these values are as follows:

SIG_DFL — signal-specific default action

> The default actions for the signals defined in this standard are specified in the preceding tables.
>
> If the default action is to stop the process, the execution of that process is temporarily suspended. When a process stops, a SIGCLD signal shall be generated for its parent process, if the parent process has set the SA_CLDSTOP flag (see *sigaction*() §3.3.4). While a process is stopped, any additional signals that are sent to the process shall not be delivered until the process is continued. An exception to this is SIGKILL, which always terminates the receiving process. Another exception is SIGCONT, which always causes the receiving process to continue. For implementations that support the Job Control Option, a process whose parent has terminated shall be sent a SIGKILL signal if the SIGTSTP, SIGTTIN, or SIGTTOU signals are generated for the process.
>
> If a signal action is set to SIG_DFL while the signal is pending, the signal shall remain pending.

SIG_IGN — ignore signal

> Delivery of the signal shall have no effect on the process.
>
> The system shall not allow the action for the signals SIGKILL, SIGSTOP, or SIGCONT to be set to SIG_IGN.

If a signal action is set to SIG_IGN while the signal is pending, the pending signal shall be discarded.

If a process sets the action for the SIGCLD signal to SIG_IGN, the behavior is implementation defined.

pointer to a function — catch signal

On delivery of the signal, the receiving process is to execute the signal-catching function at the specified address. The signal number is passed as the first argument to the signal-catching function. Other implementation specific and signal-specific arguments are allowed. After returning from the signal-catching function, the receiving process shall resume execution at the point it was interrupted.

If a signal action is set to a *pointer to a function* while the signal is pending, the signal shall remain pending.

The action taken upon normal return from a signal-catching function for signals SIGFPE, SIGILL, or SIGSEGV is implementation defined.

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

If a process establishes a signal-catching function for the SIGCLD signal while it has any child processes, the behavior is implementation defined.

If a process attempts to establish a signal-catching function for the SIGCONT signal, the behavior is implementation defined.

When signal-catching functions are invoked asynchronously with process execution, the behavior of some of the functions defined by this standard is unspecified if they are called from a signal-catching function. The following table defines a set of functions that shall be reentrant with respect to signals (that is, applications may invoke them, without restriction, from signal-catching functions):

_exit()	*access*()	*alarm*()
chdir()	*chmod*()	*chown*()
close()	*creat*()	*dup2*()
dup()	*exec*()	*fcntl*()
fork()	*fstat*()	*getegid*()
geteuid()	*getgid*()	*getgroups*()
getpgrp()	*getpid*()	*getppid*()
getuid()	*jcgetpgrp*()	*jcsetpgrp*()
kill()	*link*()	*lseek*()

mkdir()	*mkfifo*()	*open*()
pause()	*pipe*()	*read*()
rename()	*rmdir*()	*setgid*()
setpgrp()	*setuid*()	*sigaction*()
sigaddset()	*sigdelset*()	*sigfillset*()
siginitset()	*sigismember*()	*signal*()
sigpending()	*sigprocmask*()	*sigsuspend*()
sleep()	*stat*()	*tcdrain*()
tcflow()	*tcflush*()	*tcgetattr*()
tcgetpgrp()	*tcsendbreak*()	*tcsetattr*()
tcsetpgrp()	*time*()	*times*()
umask()	*uname*()	*unlink*()
ustat()	*utime*()	*wait2*()
wait()	*write*()	

All IEEE Std 1003.1 functions not in the above table and all functions defined in the *ANSI/X3.159-198x Programming Language C Standard* not stated to be callable from a signal-catching function are considered to be *unsafe* with respect to signals. If any function that is *unsafe* is interrupted by a signal-catching function that then calls any function that is *unsafe*, the behavior is undefined.

3.3.2 Send a Signal to a Process

Function: *kill*()

3.3.2.1 Synopsis

```
#include <signal.h>

int kill (pid, sig)
int pid, sig;
```

3.3.2.2 Description

The *kill*() function shall send a signal to a process or a group of processes specified by *pid*. The signal to be sent is specified by *sig* and is either one from the list given in <signal.h> §3.3.1 or zero. If *sig* is zero (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of *pid*.

For a process to have permission to send a signal to a process designated by *pid*, the real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the sending process has appropriate privileges. If both {_POSIX_KILL_SAVED} and {_POSIX_SAVED_IDS} are defined, the saved set-user-ID of the receiving process shall be checked in place of its effective user ID. If a receiving process's effective user ID has been altered through use of the S_ISUID mode bit (see

<sys/stat.h> §5.6.1), it may still receive a signal sent by the parent process or by a process with the same real user ID. The calling process may be restricted from sending a signal by implementation defined constraints.

If *pid* is greater than zero, *sig* shall be sent to the process whose process ID is equal to *pid*.

If *pid* is zero, *sig* shall be sent to all processes (excluding an implementation defined set of system processes) whose process group ID is equal to the process group ID of the sender.

If *pid* is −1, *sig* shall be sent to all processes (excluding the special set of system processes). If {_POSIX_KILL_PID_NEG1} is defined, *sig* also shall be sent to the sending process; otherwise, it shall not be sent to the sending process.

If *pid* is negative but not −1, *sig* shall be sent to all processes whose process group ID is equal to the absolute value of *pid*. The absolute value of *pid* shall not exceed {PID_MAX}.

If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* is not blocked, then either *sig* or at least one pending unblocked signal shall be delivered to the sending process before the *kill*() function returns.

As a single special case on implementations that support the Job Control Option, if the sending process has a controlling terminal, the *kill*() function shall allow the SIGCONT signal to be sent to any process that has the same controlling terminal as the sending process.

A process may be restricted from sending a signal, including the null signal, to a particular process by implementation defined constraints.

The *kill*() function is successful if the process has permission to send *sig* to any of the processes specified by *pid*. If the *kill*() function fails, no signal shall be sent.

3.3.2.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

3.3.2.4 Errors

If any of the following conditions occur, the *kill*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The value of the *sig* argument is not a valid signal number.

[EPERM] The process does not have permission to send the signal to any receiving process.

[ESRCH] No process can be found corresponding to that specified by *pid*.

3.3.2.5 References

getpid() §4.1.1, *setpgrp*() §4.3.2, *sigaction*() §3.3.4, <signal.h> §3.3.1.

3.3.3 Manipulate Signal Sets

Functions: *siginitset*(), *sigfillset*(), *sigaddset*(), *sigdelset*(), *sigismember*()

3.3.3.1 Synopsis

```
#include <signal.h>

int siginitset (set)
sigset_t *set;

int sigfillset (set)
sigset_t *set;


int sigaddset (set, signo)
sigset_t *set;
int signo;

int sigdelset (set, signo)
sigset_t *set;
int signo;

int sigismember (set, signo)
sigset_t *set;
int signo;
```

3.3.3.2 Description

The *sigsetops* primitives manipulate sets of signals. They operate on data objects addressable by the application, not on any set of signals known to the system, such as the set blocked from delivery to a process or the set pending for a process (see <signal.h> §3.3.1).

The *siginitset*() function initializes the signal set pointed to by the argument *set*, such that all signals defined in this standard are excluded. Applications shall call *siginitset*() at least once for each object of type *sigset_t* prior to any other use of that object.

The *sigfillset*() function initializes the signal set pointed to by the argument *set*, such that all signals defined in this standard are included.

The *sigaddset*() and *sigdelset*() functions respectively add and delete the individual signal specified by the value of the argument *signo* from the signal set pointed to by the argument *set*.

The *sigismember*() function tests whether the signal specified by the value of the argument *signo* is a member of the set pointed to by the argument *set*.

3.3.3.3 Returns

Upon successful completion, the *sigismember*() function returns a value of one if the specified signal is a member of the specified set, or a value of zero if it is not. Upon successful completion, the other functions return a value of zero. For all of the above functions, if an error is detected, a value of −1 is returned and *errno* is set to indicate the error.

3.3.3.4 Errors

If any of the following conditions occur, the *sigaddset*(), *sigdelset*(), and *sigismember*() functions shall return −1 and set *errno* to the corresponding value:

[EINVAL] The value of the *signo* argument is not a valid signal number.

3.3.3.5 References

sigaction() §3.3.4, <signal.h> §3.3.1, *sigpending*() §3.3.6, *sigprocmask*() §3.3.5, *sigsuspend*() §3.3.7.

3.3.4 Examine and Change Signal Action

Function: *sigaction*()

3.3.4.1 Synopsis

```
#include <signal.h>

int sigaction (sig, act, oact)
int sig;
struct sigaction *act, *oact;
```

3.3.4.2 Description

The *sigaction*() function allows the calling process to examine and/or specify the action to be taken on delivery of a specific signal. The argument *sig* specifies the signal; acceptable values are defined in <signal.h> §3.3.1.

The structure *sigaction*, used to describe an action to be taken, is defined in the header <signal.h> to include at least the following members:

Member Type	Member Name	Description
void (*)()	sa_handler	SIG_DFL, SIG_IGN, or pointer to a function
sigset_t	sa_mask	set of signals to be blocked during execution of signal-catching function
int	sa_flags	special flags to be used when delivering signal

If the argument *act* is not NULL, it points to a structure specifying the action to be taken when delivering the specified signal. If the argument *oact* is not NULL, the action previously associated with the signal is stored in the location pointed to by the argument *oact*. If the argument *act* is NULL, signal handling is unchanged; thus, the call can be used to inquire about the current handling of a given signal.

The *sa_flags* field can be used to modify the delivery of the specified signal. If *sig* is SIGCLD and the implementation supports the Job Control Option, the following flag bit, defined in the header <signal.h>, can be set in *sa_flags*:

Symbolic Constant	Description
SA_CLDSTOP	Also generate SIGCLD when children stop

An implementation may define additional flag bits in the *sa_flags* field.

When a signal is caught by a signal-catching function installed by the *sigaction*() function, a new signal mask is calculated and installed for the duration of the signal-catching function (or until a *sigprocmask*() or *sigsuspend*() function is made). This mask is formed by taking the union of the current signal mask and the set associated with the action for the signal being delivered, and then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested (by another call to the *sigaction*() function), or until one of the *exec* functions is called.

The set of signals specified by the *sa_mask* field pointed to by the argument *act* is not allowed to block those signals that cannot be ignored, as defined in <signal.h> §3.3.1. This shall be enforced by the system without causing an error to be indicated.

If the *sigaction*() function fails, no new signal handler is installed.

3.3.4.3 Returns

Upon successful completion a value of zero is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

3.3.4.4 Errors

If any of the following conditions occur, the *sigaction*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The value of the *sig* argument is not a valid signal number, or an attempt is made to supply an action for a signal that cannot be caught or ignored. See <signal.h> §3.3.1.

3.3.4.5 References

kill() §3.3.2, <signal.h> §3.3.1. *sigprocmask*() §3.3.5, *sigsetops* §3.3.3, *sigsuspend*() §3.3.7.

3.3.5 Examine and Change Blocked Signals

Function: *sigprocmask*()

3.3.5.1 Synopsis

```
#include <signal.h>

int sigprocmask (how, set, oset)
int how;
sigset_t *set, *oset;
```

3.3.5.2 Description

The *sigprocmask*() function is used to examine and/or change the calling process's signal mask. If the value of the argument *set* is not NULL, it points to a set of signals to be used to change the currently blocked set.

The value of the argument *how* indicates the manner in which the set is changed, and shall consist of one of the following values, as defined in the header <signal.h> §3.3.1:

SIG_BLOCK The resulting set shall be the union of the current set and the signal set pointed to by the argument *set*.

SIG_UNBLOCK The resulting set shall be the intersection of the current set and the complement of the signal set pointed to by the argument *set*.

SIG_SETMASK The resulting set shall be the signal set pointed to by the argument *set*.

If the argument *oset* is not NULL, the previous mask is stored in the space pointed to by *oset*. If the value of the argument *set* is NULL, the value of the argument *how* is not significant and the process's signal mask is unchanged; thus, the call can be used to enquire about currently blocked signals.

If there are any pending unblocked signals after the call to the *sigprocmask*() function, at least one of those signals shall be delivered before the *sigprocmask*() function returns.

It is not possible to block those signals that cannot be ignored, as documented in <signal.h> §3.3.1; this shall be enforced by the system without causing an error to be indicated.

If the *sigprocmask*() function fails, the process's signal mask is not changed.

3.3.5.3 Returns

Upon successful completion a value of zero is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

3.3.5.4 Errors

If any of the following conditions occur, the *sigprocmask*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The value of the *how* argument is not equal to one of the defined values.

3.3.5.5 References

sigaction() §3.3.4, <signal.h> §3.3.1. *sigpending*() §3.3.6, *sigsetops* §3.3.3, *sigsuspend*() §3.3.7.

3.3.6 Examine Pending Signals

Function: *sigpending*()

3.3.6.1 Synopsis

```
#include <signal.h>

int sigpending (set)
sigset_t *set;
```

3.3.6.2 Description

The *sigpending*() function shall store the set of signals that are blocked from delivery and pending for the calling process, in the space pointed to by the argument *set*.

3.3.6.3 Returns

Upon successful completion a value of zero is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

3.3.6.4 Errors

This standard does not specify any error conditions that are required to be detected for the *sigpending*() function. Some errors may be detected under implementation defined conditions.

3.3.6.5 References

<signal.h> §3.3.1, *sigprocmask*() §3.3.5, *sigsetops* §3.3.3.

3.3.7 Wait for a Signal

Function: *sigsuspend*()

3.3.7.1 Synopsis

```
#include <signal.h>

int sigsuspend (sigmask)
sigset_t *sigmask;
```

3.3.7.2 Description

The *sigsuspend*() function replaces the process's signal mask with the set of signals pointed to by the argument *sigmask* and then suspends the process until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

If the action is to terminate the process, the *sigsuspend*() function shall not return. If the action is to execute a signal-catching function, the *sigsuspend*() shall return after the signal-catching function returns, with the signal mask restored to the set that existed prior to the *sigsuspend*() call.

It is not possible to block those signals that cannot be ignored, as documented in <signal.h> §3.3.1; this shall be enforced by the system without causing an error to be indicated.

3.3.7.3 Returns

Since the *sigsuspend*() function suspends process execution indefinitely, there is no successful completion return value. A value of −1 is returned and *errno* is set to indicate the error.

3.3.7.4 Errors

If any of the following conditions occur, the *sigsuspend*() function shall return −1 and set *errno* to the corresponding value:

[EINTR] A signal is caught by the calling process and control is returned from the signal-catching function.

3.3.7.5 References

pause() §3.4.2, *sigaction*() §3.3.4, <signal.h> §3.3.1, *sigpending*() §3.3.6, *sigprocmask*() §3.3.5, *sigsetops* §3.3.3.

3.4 Timer Operations

A process can suspend itself for a specific period of time with the *sleep*() function or suspend itself indefinitely with the *pause*() function until a signal arrives. The *alarm*() function schedules a signal to arrive at a specific time, so a *pause*() suspension need not be indefinite.

3.4.1 Process Alarm Clock

Function: *alarm*()

3.4.1.1 Synopsis

```
unsigned int alarm (seconds)
unsigned int seconds;
```

3.4.1.2 Description

The *alarm*() function shall instruct the calling process's alarm clock to send the signal SIGALRM to the calling process after the number of real time seconds specified by *seconds* have elapsed; see *signal*().

Processor scheduling delays may cause the process to not actually begin handling the signal until after the desired time. Also, an alarm may occur up to one second early.

Alarm requests are not stacked; successive calls reset the calling process's alarm clock.

If *seconds* is 0, any previously made *alarm*() request is canceled.

3.4.1.3 Returns

The *alarm*() function shall return the amount of time remaining in the calling process's B
alarm clock from the previous *alarm*() request or zero if there is no previous *alarm*()
request.

3.4.1.4 References

exec §3.1.2, *fork*() §3.1.1, *pause*() §3.4.2, *sigaction*() §3.3.4. C

3.4.2 Suspend Process Execution

Function: *pause*()

3.4.2.1 Synopsis

```
int pause ( )
```

3.4.2.2 Description

The *pause*() function suspends the calling process until delivery of a signal whose action B
is either to execute a signal-catching function or to terminate the process. B

If the action is to terminate the process, the *pause*() function shall not return. B

If the action is to execute a signal-catching function, the *pause*() function shall return B
after the signal-catching function returns. B

3.4.2.3 Returns

Since the *pause*() function suspends process execution indefinitely, there is no successful 9
completion return value. A value of −1 is returned and *errno* is set to indicate the error. 9

3.4.2.4 Errors 9

If any of the following conditions occur, the *pause*() function shall return −1 and set B
errno to the corresponding value: B

[EINTR] A signal is caught by the calling process and control is returned 9
from the signal-catching function. 9

3.4.2.5 References

alarm() §3.4.1, *kill*() §3.3.2, *sigaction*() §3.3.4, *wait* §3.2.1. C

3.4.3 Delay Process Execution

Function: *sleep*()

3.4.3.1 Synopsis

```
unsigned int sleep (seconds)
unsigned int seconds;
```

3.4.3.2 Description

The *sleep*() function shall cause the current process to be suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons:

1. because of timer imprecision, and

2. because any caught signal shall terminate the *sleep*() function following execution of that signal's catching routine.

The suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system.

The routine shall behave as if implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal shall be saved and restored. The calling process may have set up an alarm signal before calling *sleep*(); if the *sleep*() time exceeds the time until such alarm signal, the process sleeps only until the alarm signal would have occurred, and the caller's alarm catch routine is executed just before the *sleep*() routine returns, but if the *sleep*() time is less than the time until such alarm, the prior alarm time shall go off at the same time it would have without the intervening *sleep*().

3.4.3.3 Returns

The value returned by the *sleep*() function shall be the unslept amount (the requested time minus the time actually slept). This return value may be non-zero in cases where the caller had an alarm set to go off earlier than the end of the requested time, or where *sleep*() was interrupted due to another caught signal.

3.4.3.4 References

alarm() §3.4.1, *pause*() §3.4.2, *sigaction*() §3.3.4.

4. Process Environment

4.1 Process Identification

4.1.1 Get Process and Parent Process IDs

Functions: *getpid*(), *getppid*()

4.1.1.1 Synopsis

```
int getpid ()

int getppid ()
```

4.1.1.2 Description

The *getpid*() function returns the process ID of the calling process.

The *getppid*() function returns the parent process ID of the calling process.

4.1.1.3 References

exec §3.1.2, *fork*() §3.1.1, *kill*() §3.3.2. 8

4.2 User Identification

4.2.1 Get Real User, Effective User, Real Group, and Effective Group IDs

Functions: *getuid*(), *geteuid*(), *getgid*(), *getegid*()

4.2.1.1 Synopsis

#include <sys/types.h> B

uid_t getuid () 8

uid_t geteuid () 8

uid_t getgid () 8

uid_t getegid () 8

4.2.1.2 Description

The *getuid*() function returns the real user ID of the calling process.

The *geteuid*() function returns the effective user ID of the calling process.

The *getgid*() function returns the real group ID of the calling process.

The *getegid*() function returns the effective group ID of the calling process.

4.2.1.3 References

setuid() §4.2.2.

4.2.2 Set User and Group IDs

Functions: *setuid*(), *setgid*()

4.2.2.1 Synopsis

```
#include <sys/types.h>

int setuid (uid)
uid_t uid;

int setgid (gid)
uid_t gid;
```

4.2.2.2 Description

If {_POSIX_SAVED_IDS} is defined:

If the process has appropriate privileges, the *setuid*() function sets the real user ID, effective user ID, and the saved set-user-ID to *uid*.

If the process does not have appropriate privileges, but *uid* is equal to the real user ID or the saved set-user-ID, the *setuid*() function sets the effective user ID to *uid*; the real user ID and saved set-user-ID remain unchanged.

If the process has appropriate privileges, the *setgid*() function sets the real group ID, effective group ID, and the saved set-group-ID to *gid*.

If the process does not have appropriate privileges, but *gid* is equal to the real group ID or the saved set-group-ID, the *setgid*() function sets the effective group ID to *gid*; the real group ID and saved set-group-ID remain unchanged.

Otherwise:

If the process has appropriate privileges, the *setuid*() function sets the real user ID and effective user ID to *uid*.

If the process does not have appropriate privileges, but *uid* is equal to the real user ID, the *setuid*() function sets the effective user ID to *uid*; the real user ID remains unchanged.

If the process has appropriate privileges, the *setgid*() function sets the real group ID and effective group ID to *gid*.

If the process does not have appropriate privileges, but *gid* is equal to the real group ID, the *setgid*() function sets the effective group ID to *gid*; the real group ID remains unchanged.

4.2.2.3 Returns

Upon successful completion, a value of zero is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

4.2.2.4 Errors

If any of the following conditions occur, the *setuid*() function shall return −1 and set *errno* to the corresponding value:

- [EINVAL] The value of the *uid* argument is less than zero or exceeds {UID_MAX}.
- [EPERM] The process does not have appropriate privileges and *uid* does not match the real user ID or, if {_POSIX_SAVED_IDS} is defined, the saved set-user-ID.

If any of the following conditions occur, the *setgid*() function shall return −1 and set *errno* to the corresponding value:

- [EINVAL] The value of the *gid* argument is less than zero or exceeds {UID_MAX}.
- [EPERM] The process does not have appropriate privileges and *gid* does not match the real group ID or, if {_POSIX_SAVED_IDS} is defined, the saved set-group-ID.

4.2.2.5 References

exec §3.1.2, *getuid*() §4.2.1.

4.2.3 Get Supplementary Group IDs

Function: *getgroups*()

4.2.3.1 Synopsis

```
#include <sys/types.h>

int getgroups (gidsetsize, grouplist)
int gidsetsize;
uid_t grouplist[];
```

4.2.3.2 Description

The *getgroups*() function fills in the array *grouplist* with the supplementary group IDs of the calling process. The *gidsetsize* argument gives the length of the supplied array *grouplist*. The actual number of supplementary group IDs is returned. The values of array entries with indices larger than or equal to the returned value are undefined. As a special case, if the *gidsetsize* argument is zero, *getgroups*() returns the number of supplementary group IDs associated with the calling process without modifying the array pointed to by the *grouplist* argument.

Implementation of *getgroups*() is optional on systems that have defined {NGROUPS_MAX} to be zero.

4.2.3.3 Returns

The number of supplementary group IDs is returned if successful. A return value of −1 indicates failure and *errno* is set to indicate the error.

4.2.3.4 Errors

If any of the following conditions occur, the *getgroups*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The *gidsetsize* argument is less than the number of supplementary group IDs.

4.2.3.5 References

setgid() §4.2.2.

4.2.4 Get User Name

Functions: *getlogin*(), *cuserid*()

4.2.4.1 Synopsis

```
char *getlogin ()

#include <stdio.h>

char *cuserid (s)
char *s;
```

4.2.4.2 Description

These functions return a string giving a name of the user associated with the current process. The *cuserid*() function returns a name associated with the effective user ID of the process, and the *getlogin*() function returns the name associated by the login activity with the control terminal.

The recommended procedure is either to call the *cuserid*() function, or to call *getlogin*() and, if it fails, to call the *getpwuid*() function with the value returned by the *getuid*() function.

The *getlogin*() function returns a pointer to the user's login name. The same user ID may be shared by several login names. Therefore, to ensure that the correct password database entry is found, the *getlogin*() function should be used with the *getpwnam*() function.

If *getlogin*() returns a non-NULL pointer, then that pointer is to the name the user logged in under, even if there are several login names with the same user ID.

The *cuserid*() function generates a character representation of the login name of the owner of the current process. If *s* is not a NULL pointer, it is assumed that *s* points to an array of at least **L_cuserid** characters; the representation is returned in this array. The symbolic constant **L_cuserid** is defined in <stdio.h>, and shall have a value greater than zero.

4.2.4.3 Returns

The *getlogin*() function returns a pointer to a string containing the user's login name, or a NULL pointer if the user's login name cannot be found.

If *s* is a NULL pointer, the result from *cuserid*() is generated in an area that may be static, the address of which is returned. If the login name cannot be found, *cuserid*() returns NULL. If *s* is not a NULL pointer, *s* is returned. If the login name cannot be found, the null character '\0' shall be placed at **s*.

The return value from *getlogin*() may point to static data that is overwritten by each call.

The implementation of the *cuserid*() function may use the *getpwnam*() function, so the results of a user's call to either routine may be overwritten by a subsequent call to the other routine.

4.2.4.4 Errors

This standard does not specify any error conditions that are required to be detected for the *cuserid*() function. Some errors may be detected under implementation defined conditions.

4.2.4.5 References

getpwent() §9.2.2, *getpwuid*() §9.2.2.

4.3 Process Groups

4.3.1 Get Process Group ID

Function: *getpgrp*()

4.3.1.1 Synopsis

```
int getpgrp ()
```

4.3.1.2 Description

The *getpgrp*() function returns the process group ID of the calling process.

4.3.1.3 References

setpgrp() §4.3.2, *sigaction*() §3.3.4.

4.3.2 Set Process Group ID

Function: *setpgrp*()

4.3.2.1 Synopsis

```
int setpgrp ()
```

4.3.2.2 Description

The *setpgrp*() function shall set the process group ID of the calling process to the process ID of the calling process and return the new process group ID. If the calling process is not already the process group leader, it becomes a session process group leader and releases its controlling terminal by clearing the terminal group ID.

4.3.2.3 Returns

The *setpgrp*() function returns the value of the new process group ID.

4.3.2.4 References

exec §3.1.2, *_exit*() §3.2.2, *fork*() §3.1.1, *getpid*() §4.1.1, *kill*() §3.3.2, *sigaction*() §3.3.4.

4.3.3 Set Process Group ID for Job Control

Function: *jcsetpgrp*()

4.3.3.1 Synopsis

```
int jcsetpgrp (pgrp)
int pgrp;
```

4.3.3.2 Description

This function is provided if the implementation supports the Job Control Option.

The *jcsetpgrp*() function shall set the process group ID of the calling process to *pgrp*. If *pgrp* is equal to the process ID of the calling process, the calling process becomes a job control process group leader unless the process is already the process group leader.

4.3.3.3 Returns

Upon successful completion, the *jcsetpgrp*() function returns a value of zero. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

4.3.3.4 Errors

If any of the following conditions occur, the *jcsetpgrp*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The value of the *pgrp* argument is less than or equal to zero or exceeds {PID_MAX}.

The calling process is the process group leader and the *pgrp* argument does not match the process ID.

[EPERM] The value of the *pgrp* argument is greater than zero and less than or equal to {PID_MAX} and there are processes already in the process group indicated by *pgrp* and none of these processes have the same controlling terminal as the calling process.

[ENOTTY] The calling process does not have a controlling terminal.

4.3.3.5 References

tcsetpgrp() §7.2.4.

4.4 System Identification

4.4.1 System Name

Function: *uname*()

4.4.1.1 Synopsis

```
#include <sys/utsname.h>

int uname (name)
struct utsname *name;
```

4.4.1.2 Description

The *uname*() function stores information identifying the current operating system in the structure pointed to by the argument *name*.

The structure *utsname* is defined in the header <sys/utsname.h>, and contains at least the following members:

Member Name	Description
sysname	Name of this implementation of the operating system
nodename	Name of this node within an implementation specified communications network
release	Current release level of this implementation
version	Current version level of this release
machine	Name of the hardware type that the system is running on

Each of these data items is a null-terminated character array. Additional, implementation defined, information may also be included in the structure.

The format of each member is implementation defined. The system documentation (see Documentation §2.2.1.2) shall specify the source and format of each member and may specify the range of values for each member.

4.4.1.3 Returns

Upon successful completion, a non-negative value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

4.4.1.4 Errors

This standard does not specify any error conditions that are required to be detected for the *uname*() function. Some errors may be detected under implementation defined conditions.

4.5 Time

4.5.1 Get System Time

Function: *time*()

4.5.1.1 Synopsis

```
#include <time.h>

time_t time (tloc)
time_t *tloc;
```

4.5.1.2 Description

The *time*() function returns the value of time in seconds since the Epoch (see Epoch §2.3).

If the argument *tloc* is not a NULL pointer, the return value is also stored in the location pointed to by *tloc*.

4.5.1.3 Returns

Upon successful completion, *time*() returns the value of time. Otherwise, a value of ((time_t)−1) is returned and *errno* is set to indicate the error.

4.5.1.4 Errors

This standard does not specify any error conditions that are required to be detected for the *time*() function. Some errors may be detected under implementation defined conditions.

4.5.2 Process Times

Function: *times*()

4.5.2.1 Synopsis

```
#include <sys/types.h>
#include <sys/times.h>

clock_t times (buffer)                                   8
struct tms *buffer;
```

4.5.2.2 Description

The *times*() function shall fill the structure pointed to by *buffer* with time-accounting information. The *tms* structure is defined in <sys/times.h>; it shall contain at least the following members:

Member Type	Member Name	Description	
clock_t	tms_utime	User CPU time	8
clock_t	tms_stime	System CPU time	8
clock_t	tms_cutime	User CPU time of descendants	8
clock_t	tms_cstime	System CPU time of descendants	8

All times are in {CLK_TCK}ths of a second. Additional data elements may also be declared in this structure. c

The times of a child process are included in the times of the parent when a *wait*() or *wait2*() function returns the process ID of a terminated child. See *wait* §3.2.1. If a child process has not waited for its terminated children, their times shall not be included in its times. 8

The value *tms_utime* is the CPU time used while executing instructions of the calling process. c

The value *tms_stime* is the CPU time used by the system on behalf of the calling process.

The value *tms_cutime* is the sum of the *tms_utime*s and *tms_cutime*s of the child processes.

The value *tms_cstime* is the sum of the *tms_stime*s and *tms_cstime*s of the child processes.

4.5.2.3 Returns

Upon successful completion, *times*() shall return the elapsed real time, in {CLK_TCK}ths of a second, since an arbitrary point in the past (for example, system start-up time). This point does not change from one invocation of *times*() within the process to another. The return value may overflow the possible range of type *clock_t*. If the *times*() function fails, a value of ((clock_t) −1) is returned and *errno* is set to indicate the error.

4.5.2.4 References

exec §3.1.2, *fork*() §3.1.1, *time*() §4.5.1, *wait*() §3.2.1.

4.6 Environment Variables

4.6.1 Environment Access

Function: *getenv*()

4.6.1.1 Synopsis

```
char *getenv (name)
char *name;
```

4.6.1.2 Description

The *getenv*() function searches the environment list (see **Environment Description** §2.7) for a string of the form *name=value* and returns a pointer to *value* if such a string is present. If the specified *name* cannot be found, a NULL pointer is returned.

4.6.1.3 Errors

This standard does not specify any error conditions that are required to be detected for the *getenv*() function. Some errors may be detected under implementation defined conditions.

4.6.1.4 References

environ §3.1.2, **Environment Description** §2.7.

4.7 Terminal Identification

4.7.1 Generate Terminal Pathname

Function: *ctermid*()

4.7.1.1 Synopsis

```
#include <stdio.h>

char *ctermid (s)
char *s;
```

4.7.1.2 Description

The *ctermid*() function generates a string that, when used as a pathname, refers to the controlling terminal for the current process.

If the *ctermid*() function returns a pathname, access to the file is not guaranteed.

4.7.1.3 Returns

If *s* is a NULL pointer, the string is stored in an internal static area, the contents of which may be overwritten at the next call to *ctermid*(), and the address of which is returned; otherwise *s* is assumed to point to a character array of at least **L_ctermid** elements; the string is placed in this array and the value of *s* is returned. The symbolic constant **L_ctermid** is defined in **<stdio.h>**, and shall have a value greater than zero.

The *ctermid*() function shall return an empty string if the pathname for the controlling terminal cannot be determined.

4.7.1.4 Errors

This standard does not specify any error conditions that are required to be detected for the *ctermid*() function. Some errors may be detected under implementation defined conditions.

4.7.1.5 References

ttyname() §4.7.2.

4.7.2 Determine Terminal Device Name

Functions: *ttyname*(), *isatty*()

4.7.2.1 Synopsis

```
char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;
```

4.7.2.2 Description

The *ttyname*() function returns a pointer to a string containing a null-terminated pathname of the terminal associated with file descriptor *fildes*.

The return value of *ttyname*() may point to static data that is overwritten by each call.

The *isatty*() function returns 1 if *fildes* is a valid file descriptor associated with a terminal, zero otherwise.

4.7.2.3 Returns

The *ttyname*() function returns a NULL pointer if *fildes* is not a valid file descriptor associated with a terminal device.

4.7.2.4 Errors

This standard does not specify any error conditions that are required to be detected for the *ttyname*() function. Some errors may be detected under implementation defined conditions.

4.8 Configurable System Variables

4.8.1 Get Configurable System Variables

Function: *sysconf*()

4.8.1.1 Synopsis

```
#include <unistd.h>

long sysconf (name)
int name;
```

4.8.1.2 Description

The *sysconf*() function provides a method for the application to determine the current value of a configurable system limit or option (*variable*).

The *name* argument represents the system variable to be queried. The following table lists the system variables from **<limits.h>** §2.9 or **<unistd.h>** §2.10 that can be returned by *sysconf*(), and the symbolic constants, defined in **<unistd.h>**, that are the corresponding values used for *name*:

Variable	*name* Value
ARG_MAX	_SC_ARG_MAX
CHILD_MAX	_SC_CHILD_MAX
CLK_TCK	_SC_CLK_TCK
NGROUPS_MAX	_SC_NGROUPS_MAX
OPEN_MAX	_SC_OPEN_MAX
PASS_MAX	_SC_PASS_MAX
PID_MAX	_SC_PID_MAX
UID_MAX	_SC_UID_MAX
_POSIX_EXIT_SIGHUP	_SC_EXIT_SIGHUP
_POSIX_JOB_CONTROL	_SC_JOB_CONTROL
_POSIX_KILL_PID_NEG1	_SC_KILL_PID_NEG1
_POSIX_KILL_SAVED	_SC_KILL_SAVED
_POSIX_PGID_CLEAR	_SC_PGID_CLEAR
_POSIX_SAVED_IDS	_SC_SAVED_IDS
_POSIX_VERSION	_SC_VERSION

4.8.1.3 Returns

If the variable corresponding to *name* is not defined on the system, or if *name* is an invalid value, the *sysconf*() function returns −1.

Otherwise, the *sysconf*() function returns the current variable value on the system. The value returned shall not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's **<limits.h>** §2.9 or **<unistd.h>** §2.10. The value shall not change during the lifetime of the calling process.

5. Files and Directories

The functions in this section perform the operating system services dealing with the creation and removal of files and directories and the detection and modification of their characteristics. They also provide the primary methods a process will use to gain access to files and directories for subsequent I/O operations (see **Input and Output Primitives** §6).

5.1 Directories

5.1.1 Format of Directory Entries

5.1.1.1 Synopsis

```
#include <sys/types.h>
#include <dirent.h>
```

5.1.1.2 Description

The header **<dirent.h>** defines a structure and a defined type used by the *directory* routines.

The internal format of directories is implementation defined.

The routine *readdir*() returns a pointer to an object of type *struct dirent* that includes the member:

Member Type	Member Name	Description
char [ ]	d_name	Null-terminated filename

The character array *d_name* is of unspecified size, but the number of characters preceding the terminating null character shall not exceed {NAME_MAX}.

Additional, implementation defined, structure elements may also be declared in this structure by the header **<dirent.h>**.

5.1.1.3 References

directory §5.1.2.

5.1.2 Directory Operations

Functions: *opendir*(), *readdir*(), *rewinddir*(), *closedir*()

5.1.2.1 Synopsis

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir (dirname)
char *dirname;

struct dirent *readdir (dirp)
DIR *dirp;

void rewinddir (dirp)
DIR *dirp;

int closedir (dirp)
DIR *dirp;
```

5.1.2.2 Description

The type DIR, which is defined in the header <dirent.h> §5.1.1, represents a *directory stream*, which is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files; files may be removed from a directory or added to a directory asynchronous to the operations described in this section.

The *opendir*() function opens a directory stream corresponding to the directory named by the *dirname* argument. The directory stream is positioned at the first entry.

If a file is removed from or added to the directory after the most recent call to *opendir*() or *rewinddir*(), whether a subsequent call to *readdir*() returns an entry for that file is unspecified.

The *readdir*() function returns a pointer to a structure representing the directory entry at the current position in the directory stream to which *dirp* refers, and positions the directory stream at the next entry. It returns a NULL pointer upon reaching the end of the directory stream.

The *readdir*() function shall not return directory entries containing empty names. If {_POSIX_DIR_DOTS} is in effect for *dirname*, entries for dot or dot-dot shall be returned; otherwise they shall not be returned.

The pointer returned by *readdir*() points to data which may be overwritten by another call to *readdir*() on the same directory stream. This data shall not be overwritten by another call to *readdir*() on a different directory stream.

Upon successful completion, the *readdir*() function shall mark for update the *st_atime* field of the directory.

The *rewinddir*() function resets the position of the directory stream to which *dirp* refers to the beginning of the directory. It also causes the directory stream to refer to the current state of the corresponding directory, as a call to *opendir*() would have done. It does not return a value. If *dirp* does not refer to a directory stream, the effect is undefined.

The *closedir*() function closes the directory stream referred to by *dirp* and returns a value of zero if successful. Otherwise, it returns −1 indicating an error. Upon return, the value of *dirp* may no longer point to an accessible object of type DIR.

5.1.2.3 Returns

Upon successful completion, *opendir*() returns a pointer to an object of type DIR. Otherwise, a value of NULL is returned and *errno* is set to indicate the error.

Upon successful completion, *readdir*() returns a pointer to an object of type *struct dirent*. When an error is encountered, a value of NULL is returned and *errno* is set to indicate the error. When the end of the directory is encountered, a value of NULL is returned and *errno* is not changed.

Upon successful completion, *closedir*() returns a value of zero. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

5.1.2.4 Errors

If any of the following conditions occur, the *opendir*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied for any component of *dirname* or read permission is denied for *dirname*.

[EMFILE] Too many file descriptors are currently open for the process.

[ENOTDIR] A component of *dirname* is not a directory.

For each of the following conditions, if the condition is detected, the *readdir*() function shall return −1 and set *errno* to the corresponding value:

[EBADF] The *dirp* argument does not refer to an open directory stream.

For each of the following conditions, if the condition is detected, the *closedir*() function shall return −1 and set *errno* to the corresponding value:

[EBADF] The *dirp* argument does not refer to an open directory stream.

5.1.2.5 References

<dirent.h> §5.1.1.

5.2 Working Directory

5.2.1 Change Current Working Directory

Function: *chdir*()

5.2.1.1 Synopsis

```
int chdir (path)
char *path;
```

5.2.1.2 Description

The *path* argument points to the pathname of a directory. The *chdir*() function causes the named directory to become the current working directory, that is, the starting point for path searches of pathnames not beginning with slash.

If the *chdir*() function fails, the current working directory shall remain unchanged.

5.2.1.3 Returns

Upon successful completion, a value of zero is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

5.2.1.4 Errors

If any of the following conditions occur, the *chdir*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied for any component of the pathname.

[ENAMETOOLONG]
The *path* argument exceeds {PATH_MAX} in length, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOTDIR] A component of the pathname is not a directory.

[ENOENT] The named directory does not exist or *path* is an empty string.

5.2.1.5 References

getcwd() §5.2.2.

5.2.2 Working Directory Pathname

Function: *getcwd*()

5.2.2.1 Synopsis

```
char *getcwd (buf, size)
char *buf;
int size;
```

5.2.2.2 Description

The routine *getcwd*() copies the absolute pathname of the current working directory to the character array pointed to by the argument *buf* and returns a pointer to the result. The *size* argument is the size in bytes of the character array pointed to by the *buf* argument. If *buf* is a NULL pointer, the behavior of *getcwd*() is undefined.

5.2.2.3 Returns

If successful, the *buf* argument is returned. A NULL pointer is returned if an error occurs and the variable *errno* is set to indicate the error. The contents of *buf* after an error is undefined.

5.2.2.4 Errors

If any of the following conditions occur, the *getcwd*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The *size* argument is less than or equal to zero.

[ERANGE] The *size* argument is greater than zero, but is smaller than the length of the pathname.

For each of the following conditions, if the condition is detected, the *getcwd*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] Read or search permission was denied for a component of the pathname.

5.2.2.5 References

chdir() §5.2.1.

5.3 General File Creation

5.3.1 Open a File

Function: *open*()

5.3.1.1 Synopsis

```
#include <sys/types.h>

#include <fcntl.h>

int open (path, oflag, ...)
char *path;
int oflag;
```

5.3.1.2 Description

The *open*() function establishes the connection between a file and a file descriptor. It creates an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The *path* argument points to a pathname naming a file.

The *open*() function shall return a file descriptor for the named file which is the lowest file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor does not share it with any other process in the system. The file status flags and file access modes of the open file description shall be set according to the value of *oflag*. The value of *oflag* is the bitwise inclusive OR of values from the following list. See <fcntl.h> §6.5.1 for the definitions of the symbolic constants. Implementations may define additional flags, whose names shall begin with "O_." Applications shall specify exactly one of the first three values (file access modes) below in the value of *oflag*:

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

Any combination of the remaining flags may be specified in the value of *oflag*:

O_APPEND If set, the file offset shall be set to the end of the file prior to each write.

O_CREAT This option requires a third argument, *mode*, which is of type *mode_t*. If the file exists, this flag has no effect. Otherwise, the file is created; the file's user ID shall be set to the process's effective user ID; if {_POSIX_GROUP_PARENT} is in effect for *path*, the file's group ID shall be set to the group ID of the directory in which the file is being created; otherwise, the file's group ID shall be set to the process's effective group ID. The file permission bits (see <sys/stat.h> §5.6.1) shall be set to the value of *mode* except those set in the process's file mode creation mask (see *umask*() §5.3.3). When bits in *mode* other than the file permission bits are set, the effect is implementation defined. The *mode* argument does not affect whether the file is opened for reading, for writing, or for both.

O_EXCL If O_EXCL and O_CREAT are set, *open*() shall fail if the file exists. If O_EXCL is set and O_CREAT is not set, the result is implementation defined.

O_NONBLOCK

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NONBLOCK is set:
An *open*() for reading-only shall return without delay. An *open*() for writing-only shall return an error if no process currently has the file open for reading.

If O_NONBLOCK is clear:
An *open*() for reading-only shall block until a process opens the file for writing.. An *open*() for writing-only shall block until a process opens the file for reading.

When opening a block special or character special file that supports nonblocking opens:

If O_NONBLOCK is set:
The *open*() shall return without waiting for the device to be ready or available. Subsequent behavior of the device is device specific.

If O_NONBLOCK is clear:
The *open*() shall wait until the device is ready or available before returning.

Otherwise, the behavior of O_NONBLOCK is unspecified.

O_TRUNC If the file exists and is a regular file, it shall be truncated to zero length and the mode and owner shall be unchanged.

If O_CREAT is set and the file did not previously exist, upon successful completion, the *open*() function shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file and the *st_ctime* and *st_mtime* fields of the parent directory.

If O_TRUNC is set and the file did previously exist, upon successful completion, the *open*() function shall mark for update the *st_ctime* and *st_mtime* fields of the file.

5.3.1.3 Returns

Upon successful completion, the function shall open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, it shall return −1 and shall set *errno* to indicate the error. No files shall be created or modified if the function returns −1.

5.3.1.4 Errors

If any of the following conditions occur, the *open*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *oflag* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.

[EEXIST] O_CREAT and O_EXCL are set, and the named file exists.

[EINTR] The *open*() operation was terminated prematurely by a signal.

[EISDIR] The named file is a directory and the *oflag* argument specifies write or read/write access.

[EMFILE] Too many file descriptors are currently in use by this process.

[ENAMETOOLONG]
The length of the *path* string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENFILE] Too many files are currently open in the system.

[ENOENT] O_CREAT is not set and the named file does not exist; or O_CREAT is set and either the path prefix does not exist or the *path* argument points to an empty string.

[ENOSPC] The directory or file system which would contain the new file cannot be extended.

[ENOTDIR] A component of the path prefix is not a directory.

[ENXIO] O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.

[EROFS] The named file resides on a read-only file system and either O_WRONLY, O_RDWR, or O_CREAT (if file does not exist) is set in the *oflag* argument.

5.3.1.5 References

close() §6.3.1, *creat*() §5.3.2, *dup*() §6.2.1, *exec* §3.1.2, *fcntl*() §6.5.2, <fcntl.h> §6.5.1, *lseek*() §6.5.3, *read*() §6.4.1, *sigaction*() §3.3.4, *stat*() §5.6.2, <sys/stat.h> §5.6.1, *write*() §6.4.2, *umask*() §5.3.3.

5.3.2 Create a New File or Rewrite an Existing One

Function: *creat*()

5.3.2.1 Synopsis

```
#include <sys/types.h>

int creat (path, mode)
char *path;
mode_t mode;
```

5.3.2.2 Description

The function call

```
creat (path, mode);
```

is equivalent to

```
open (path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

5.3.2.3 References

open() §5.3.1, <sys/stat.h> §5.6.1.

5.3.3 Set File Creation Mask

Function: *umask*()

5.3.3.1 Synopsis

```
#include <sys/types.h>

mode_t umask (cmask)
mode_t cmask;
```

5.3.3.2 Description

The *umask*() routine sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the file permission bits (see <sys/stat.h> §5.6.1) of *cmask* are used.

The process's file mode creation mask is used during *open*(), *creat*(), *mkdir*(), and *mkfifo*() functions to turn off permission bits in the *mode* argument supplied. Bit positions that are set in *cmask* are cleared in the mode of the created file.

5.3.3.3 Returns

The previous value of the file mode creation mask is returned.

5.3.3.4 References

chmod() §5.6.4, *creat*() §5.3.2, *mkdir*() §5.4.1, *mkfifo*() §5.4.2, *open*() §5.3.1, <signal.h> §3.3.1, <sys/stat.h> §5.6.1.

5.3.4 Link to a File

Function: *link*()

5.3.4.1 Synopsis

```
int link (path1, path2)
char *path1, *path2;
```

5.3.4.2 Description

The argument *path1* points to a pathname naming an existing file. The argument *path2* points to a pathname naming the new directory entry to be created. The *link*() function shall create a new link for the existing file. The link count of the file is incremented by one.

If the *link*() function fails, no link shall be created.

If *path1* names a directory, the effect of this function is dependent on the definition of {_POSIX_LINK_DIR}. If in effect for *path1*, the link is created, subject to any other restrictions listed for the function; otherwise, the linking of a directory shall be disallowed and the function shall fail.

Upon successful completion, the *link*() function shall mark for update the *st_ctime* field of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry are marked for update.

5.3.4.3 Returns

Upon successful completion, *link*() shall return a value of zero. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

5.3.4.4 Errors

If any of the following conditions occur, the *link*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] A component of either path prefix denies search permission, or the requested link requires writing in a directory with a mode that denies write permission.

[EEXIST] The link named by *path2* exists.

[EMLINK] The number of links to the file named by *path1* would exceed {LINK_MAX}.

[ENAMETOOLONG]
The length of the *path1* or *path2* string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOENT] A component of either path prefix does not exist; the file named by *path1* does not exist; or either *path1* or *path2* points to an empty string.

[ENOSPC] The directory that would contain the link cannot be extended.

[ENOTDIR] A component of either path prefix is not a directory.

[EPERM] The file named by *path1* is a directory and the implementation restricts the linking of directories to processes with appropriate privileges, and the calling process does not have appropriate privileges.

[EROFS] The requested link requires writing in a directory on a read-only file system.

[EXDEV] The link named by *path2* and the file named by *path1* are on different file systems and the implementation does not support links between file systems.

5.3.4.5 References

rename() §5.5.3, *unlink*() §5.5.1.

5.4 Special File Creation

5.4.1 Make a Directory

Function: *mkdir*()

5.4.1.1 Synopsis

```
#include <sys/types.h>

int mkdir (path, mode)
char *path;
mode_t mode;
```

5.4.1.2 Description

The *mkdir*() routine creates a new directory with name *path*. The file permission bits of the new directory are initialized from *mode*. The file permission bits of the *mode* argument are modified by the process's file creation mask (see *umask*() §5.3.3). When bits in *mode* other than the file permission bits are set, the effect is implementation defined.

The directory's owner ID is set to the process's effective user ID. If {_POSIX_GROUP_PARENT} is in effect for *path*, the directory's group ID shall be set to the group ID of the directory in which the directory is being created; otherwise, the directory's group ID shall be set to the process's effective group ID.

If {_POSIX_DIR_DOTS} is in effect for *path*, the newly created directory shall contain only entries for dot and dot-dot; otherwise the directory shall be empty.

Upon successful completion, the *mkdir*() function shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the directory. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry are marked for update.

5.4.1.3 Returns

A return value of zero indicates success. A return value of −1 indicates that an error has occurred and an error code is stored in *errno*. No directory shall be created if the return value is −1.

5.4.1.4 Errors

If any of the following conditions occur, the *mkdir*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be created.

[EEXIST] The named file exists.

[EMLINK] The link count of the parent directory would exceed {LINK_MAX}.

[ENAMETOOLONG]
The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOENT] A component of the path prefix does not exist or the *path* argument points to an empty string.

[ENOSPC] The file system does not contain enough space to hold the contents of the new directory or to extend the parent directory of the new directory.

[ENOTDIR] A component of the path prefix is not a directory.

[EROFS] The path prefix resides on a read-only file system.

5.4.1.5 References

chmod() §5.6.4, *stat*() §5.6.2, <sys/stat.h> §5.6.1, *umask*() §5.3.3.

5.4.2 Make a FIFO Special File

Function: *mkfifo*()

5.4.2.1 Synopsis

```
#include <sys/types.h>

int mkfifo (path, mode)
char *path;
mode_t mode;
```

5.4.2.2 Description

The *mkfifo*() routine creates a new FIFO special file named by the pathname pointed to by *path*. The mode of the new FIFO is initialized from *mode*. The file permission bits of the *mode* argument are modified by the process's file creation mask (see *umask*() §5.3.3). When bits in *mode* other than the file permission bits are set, the effect is implementation defined.

The FIFO's owner ID shall be set to the process's effective user ID. If {_POSIX_GROUP_PARENT} is in effect for *path*, the FIFO's group ID shall be set to the group ID of the directory in which the FIFO is being created; otherwise, the FIFO's group ID shall be set to the process's effective group ID.

Upon successful completion, the *mkfifo*() function shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry are marked for update.

5.4.2.3 Returns

Upon successful completion a value of zero is returned. Otherwise, a value of −1 is returned, no FIFO is created, and *errno* is set to indicate the error.

5.4.2.4 Errors

If any of the following conditions occur, the *mkfifo*() function shall return −1 and set *errno* to the corresponding value:

- [EACCES] A component of the path prefix denies search permission.
- [EEXIST] The named file already exists.
- [ENAMETOOLONG] The length of the *path* string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
- [ENOENT] A component of the path prefix does not exist or the *path* argument points to an empty string.
- [ENOSPC] The directory that would contain the new file cannot be extended or the file system is out of file allocation resources.
- [ENOTDIR] A component of the path prefix is not a directory.
- [EROFS] The named file resides on a read-only file system.

5.4.2.5 References

chmod() §5.6.4, *exec* §3.1.2, *pipe*() §6.1.1, *stat*() §5.6.2, <sys/stat.h> §5.6.1, *umask*() §5.3.3.

5.5 File Removal

5.5.1 Remove Directory Entries

Function: *unlink*()

5.5.1.1 Synopsis

```
int unlink (path)
char *path;
```

5.5.1.2 Description

The *unlink*() function shall remove the link named by the pathname pointed to by *path* and decrement the link count of the file referenced by the link.

When the file's link count becomes zero and no process has the file open, the space occupied by the file shall be freed and the file shall no longer be accessible. If one or more processes have the file open when the last link is removed, the removal shall be postponed until all references to the file have been closed.

If *path* names a directory, the effect of this function is dependent on the definition of {_POSIX_LINK_DIR}. If in effect for *path*, the link is removed, subject to any other restrictions listed for the function; otherwise, the unlinking of a directory shall be disallowed and the function shall fail. Applications should use *rmdir*() to remove a directory.

Upon successful completion, the *unlink*() function shall mark for update the *st_ctime* and *st_mtime* fields of the parent directory. Also, if the file's link count is not zero, the *st_ctime* field of the file shall be marked for update.

5.5.1.3 Returns

Upon successful completion, a value of zero shall be returned. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error. If −1 is returned, the named file shall not be changed.

5.5.1.4 Errors

If any of the following conditions occur, the *unlink*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied for a component of the path prefix, or write permission is denied on the directory containing the link to be removed.

[ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOENT] The named file does not exist or the *path* argument points to an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

[EROFS] The directory entry to be unlinked is part of a read-only file system.

For each of the following conditions, if the condition is detected, the *unlink*() function shall return −1 and set *errno* to the corresponding value:

[EBUSY] The file named by the *path* argument cannot be unlinked because it is being used by the system or another process.

[EPERM] The named file is a directory and the implementation restricts the unlinking of directories to processes with appropriate privileges, and the calling process does not have appropriate privileges.

5.5.1.5 References

close() §6.3.1, *link*() §5.3.4, *open*() §5.3.1, *rename*() §5.5.3, *rmdir*() §5.5.2.

5.5.2 Remove a Directory

Function: *rmdir*()

5.5.2.1 Synopsis

```
int rmdir (path)
char *path;
```

5.5.2.2 Description

The *rmdir*() function removes a directory whose name is given by *path*. If {_POSIX_DIR_DOTS} is in effect for *path*, the directory shall be removed only if there are no entries other than dot or dot-dot; otherwise the directory shall be removed only if it has no entries.

If the directory is the root directory or the current working directory, the effect of this function is implementation defined.

Upon successful completion, the *rmdir*() function shall mark for update the *st_ctime* and *st_mtime* fields of the parent directory.

5.5.2.3 Returns

A return value of zero indicates success. A return value of −1 indicates that an error has occurred and an error code has been stored in *errno*.

5.5.2.4 Errors

If any of the following conditions occur, the *rmdir*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied on a component of the path or write permission is denied on the parent directory of the directory to be removed.

[EEXIST] or [ENOTEMPTY]
The *path* argument names a directory containing files other than dot and dot-dot.

[ENAMETOOLONG]
The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOENT] The *path* argument names a non-existent directory or points to an empty string.

[ENOTDIR] A component of the path is not a directory.

[EROFS] The directory entry to be removed resides on a read-only file system.

For each of the following conditions, if the condition is detected, the *rmdir*() function shall return −1 and set *errno* to the corresponding value:

[EBUSY] The directory to be removed is currently in use by the system or another process.

5.5.2.5 References

mkdir() §5.4.1, *unlink*() §5.5.1.

5.5.3 Rename a File

Function: *rename*()

5.5.3.1 Synopsis

```
int rename (old, new)
char *old;
char *new;
```

5.5.3.2 Description

The *rename*() function changes the name of a file. The *old* argument points to the pathname of the file to be renamed. The *new* argument points to the new pathname of the file.

If the *old* argument and the *new* argument both refer to links to the same existing file, the *rename*() function shall return successfully and perform no other action.

If the *old* argument points to the pathname of a file that is not a directory, the *new* argument shall not point to the pathname of a directory. If the link named by the *new* argument exists, it shall be removed and *old* renamed to *new*. In this case, implementations shall ensure that a link named *new* remains visible to other processes throughout the renaming operation. Write access permission is required for both the directory containing *old* and the directory containing *new*.

If the *old* argument points to the pathname of a directory, the *new* argument shall not point to the pathname of a file that is not a directory. If the directory named by the *new* argument exists, it shall be removed and *old* renamed to *new*. In this case, implementations shall ensure that a link named *new* remains visible to other processes throughout the renaming operation. Thus, if *new* names an existing directory, the directory shall be required to have only the entries dot and dot-dot, if {_POSIX_DIR_DOTS} is in effect for *new*; if {_POSIX_DIR_DOTS} is not in effect, the existing directory shall be required to be empty. The *new* pathname shall not name a descendant of *old*. Write access permission is required for the directory containing *old* and the directory containing *new*. If the *old* argument points to the pathname of a directory, write access permission may be required for the directory named by *old*, and, if it exists, the directory named by *new*.

Upon successful completion, the *rename*() function shall mark for update the *st_ctime* and *st_mtime* fields of the parent directory of each file.

5.5.3.3 Returns

A return value of zero indicates success. A return value of −1 indicates that an error has occurred and an error code has been stored in *errno*.

5.5.3.4 Errors

If any of the following conditions occur, the *rename*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] A component of either path prefix denies search permission; or one of the directories containing *old* or *new* denies write permissions; or, write permission is required and is denied for a directory pointed to by the *old* or *new* arguments.

[EEXIST] or [ENOTEMPTY]
The link named by *new* is a directory containing entries other than dot and dot-dot.

[EINVAL] The *new* directory is an ancestor or a descendant of the *old* directory.

[EISDIR] The *new* argument points to a directory and the *old* argument points to a file that is not a directory.

[ENAMETOOLONG]
The length of the *old* or *new* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOENT] The link named by the *old* argument does not exist or either *old* or *new* points to an empty string.

[ENOSPC] The directory that would contain *new* cannot be extended.

[ENOTDIR] A component of either path prefix is not a directory; or the *old* argument names a directory and the *new* argument names a nondirectory file.

[EROFS] The requested operation requires writing in a directory on a read-only file system.

For each of the following conditions, if the condition is detected, the *rename*() function shall return −1 and set *errno* to the corresponding value:

[EBUSY] The link named by *old* or *new* is currently in use by the system or another process.

[EXDEV] The links named by *new* and *old* are on different file systems.

5.5.3.5 References

link() §5.3.4, *rmdir*() §5.5.2, *unlink*() §5.5.1.

5.6 File Characteristics

5.6.1 File Characteristics: Header File and Data Structure

5.6.1.1 Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
```

5.6.1.2 Description

The header <sys/stat.h> defines the structure *stat* returned by the functions *stat*() and *fstat*().

Member Type	Member Name	Description
mode_t	st_mode	File mode (see list below)
ino_t	st_ino	File serial number
dev_t	st_dev	ID of device containing a directory entry for this file. File serial number and device ID taken together uniquely identify the file within the system.
dev_t	st_rdev	ID of device. This entry is valid only for character special or block special files.
nlink_t	st_nlink	Number of links
uid_t	st_uid	User ID of the file's owner
uid_t	st_gid	Group ID of the file's group
off_t	st_size	For regular files, this is the file size in bytes. For other file types, the use of this field is unspecified.
time_t	st_atime	Time of last access
time_t	st_mtime	Time of last data modification
time_t	st_ctime	Time of last file status change

All of the described members must appear in the *stat* structure. The *stat* structure may also include other data elements as well. The structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, and *st_mtime* shall have meaningful values for all file types defined in this standard. The value of the member *st_rdev* is implementation defined. The value of the member *st_nlink* shall be set to the number of links to the file.

5.6.1.2.1 <sys/stat.h> File Types

The following macros shall test whether a file is of the specified type. The value *m* supplied to the macros is the value of *st_mode* from a *struct stat*. The macro evaluates to a non-zero value if the test is true, zero if the test is false.

S_ISDIR(*m*)	Test macro for directory file
S_ISCHR(*m*)	Test macro for character special file
S_ISBLK(*m*)	Test macro for block special file
S_ISREG(*m*)	Test macro for regular file
S_ISFIFO(*m*)	Test macro for FIFO special file

5.6.1.2.2 <sys/stat.h> File Modes

The *st_mode* value is bit-encoded with the following masks and bits:

S_IRWXU	Read, write, search (if a directory), or execute (otherwise) permissions mask for the file owner class.	
	S_IRUSR	Read permission bit for the file owner class.
	S_IWUSR	Write permission bit for the file owner class.
	S_IXUSR	Search (if a directory) or execute (otherwise) permissions bit for the file owner class.
S_IRWXG	Read, write, search (if a directory), or execute (otherwise) permissions mask for the file group class.	
	S_IRGRP	Read permission bit for the file group class.
	S_IWGRP	Write permission bit for the file group class.
	S_IXGRP	Search (if a directory) or execute (otherwise) permissions bit for the file group class.
S_IRWXO	Read, write, search (if a directory), or execute (otherwise) permissions mask for the file other class.	
	S_IROTH	Read permission bit for the file other class.
	S_IWOTH	Write permission bit for the file other class.
	S_IXOTH	Search (if a directory) or execute (otherwise) permissions bit for the file other class.
S_ISUID	Set user ID on execution. The process's effective user ID shall be set to that of the owner of the file when the file is run as a program (see *exec*). This bit should be cleared on any write to the file.	
S_ISGID	Set group ID on execution. Set effective group ID on the process to the file's group when the file is run as a program (see *exec*). This bit should be cleared on any write to the file.	

The file permission bits are defined to be those corresponding to the bitwise inclusive OR of S_IRWXU, S_IRWXG, and S_IRWXO.

5.6.1.2.3 <sys/stat.h> Time Entries

The time-related fields of *struct stat* are as follows:

st_atime	Accessed file data, e.g. *read*().
st_mtime	Modified file data, e.g. *write*().
st_ctime	Changed file status, e.g. *chmod*().

These times are updated as described by file times update §2.4.

All the functions in this standard that change these fields directly describe those changes in the context of the functions' definitions. Other functions that directly change *st_atime*, *st_mtime*, or *st_ctime* shall be implementation defined.

Times are given in seconds since the Epoch (see Epoch §2.3).

5.6.1.3 References

chmod() §5.6.4, *chown*() §5.6.5, *creat*() §5.3.2, *link*() §5.3.4, *mkdir*() §5.4.1, *mkfifo*() §5.4.2, *pipe*() §6.1.1, *read*() §6.4.1, *unlink*() §5.5.1, *utime*() §5.6.6, *write*() §6.4.2, *remove*() (*ANSI/X3.159-198x Programming Language C Standard*).

5.6.2 Get File Status

Functions: *stat*(), *fstat*()

5.6.2.1 Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

5.6.2.2 Description

The *path* argument points to a pathname naming a file. Read, write or execute permission for the named file is not required, but all directories listed in the pathname leading to the file must be searchable. The *stat*() function obtains information about the named file and writes it to the area pointed to by the *buf* argument.

Similarly, the *fstat*() function obtains information about an open file known by the file descriptor *fildes*.

Additional implementation defined access constraints may cause the *stat*() and *fstat*() functions to fail.

Both functions update any time-related fields as described in file times update §2.4 before writing into the *stat* structure.

The *buf* is taken to be a pointer to a *stat* structure, as defined in the header <sys/stat.h> §5.6.1, into which information is placed concerning the file.

5.6.2.3 Returns

Upon successful completion a value of zero shall be returned. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

5.6.2.4 Errors

If any of the following conditions occur, the *stat*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied for a component of the path prefix.

[ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOENT] The named file does not exist or the *path* argument points to an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

If any of the following conditions occur, the *fstat*() function shall return −1 and set *errno* to the corresponding value:

[EBADF] The *fildes* argument is not a valid file descriptor.

5.6.2.5 References

creat() §5.3.2, *dup*() §6.2.1, *fcntl*() §6.5.2, *open*() §5.3.1, *pipe*() §6.1.1, <sys/stat.h> §5.6.1.

5.6.3 File Accessibility

Function: *access*()

5.6.3.1 Synopsis

```
#include <unistd.h>

int access (path, amode)
char *path;
int amode;
```

5.6.3.2 Description

The *access*() function checks the accessibility of the file named by the pathname pointed to by the *path* argument for the file access permissions indicated by *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID.

The value of *amode* is either the bitwise inclusive OR of the access permissions to be checked (R_OK, W_OK, and X_OK) or the existence test, F_OK. See Symbolic Constants §2.10 for the description of these symbolic constants.

If any access permission is to be checked, each shall be checked individually, as described in file access permissions §2.4. If the process has appropriate privileges, an implementation may substitute search permissions for execute permission.

5.6.3.3 Returns

If the requested access is permitted, a value of zero shall be returned. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

5.6.3.4 Errors

If any of the following conditions occur, the *access*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] The permissions specified by *amode* are denied, or search permission is denied on a component of the path prefix.

[ENAMETOOLONG]
The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOENT] The *path* argument points to an empty string or to the name of a file that does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[EROFS] Write access requested for a file on a read-only file system.

For each of the following conditions, if the condition is detected, the *access*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] Invalid value specified for *amode*.

5.6.3.5 References

chmod() §5.6.4, *stat*() §5.6.2, <unistd.h> §2.10.

5.6.4 Change File Modes

Function: *chmod*()

5.6.4.1 Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod (path, mode)
char *path;
mode_t mode;
```

5.6.4.2 Description

The *path* argument shall point to a pathname naming a file. If the effective user ID of the calling process matches the file owner or has appropriate privileges, the *chmod*() function shall set the file mode, as described in <sys/stat.h> §5.6.1, of the named file from the corresponding bits in the *mode* argument. These bits define access permissions for the user associated with the file, the group associated with the file, and all others, as described in file access permissions §2.4. Additional implementation defined restrictions may cause the S_ISUID and S_ISGID bits in *mode* to be ignored.

If the calling process does not have appropriate privileges, and if the group ID of the file does not match the effective group ID or one of the supplementary group IDs, bit S_ISGID (set group ID on execution) in the file's mode shall be cleared upon successful return from *chmod*().

The effect on file descriptors for files open at the time of the *chmod*() function is implementation defined.

Upon successful completion, the *chmod*() function shall mark for update the *st_ctime* field of the file.

5.6.4.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error. If −1 is returned, no change to the file mode shall have occurred.

5.6.4.4 Errors

If any of the following conditions occur, the *chmod*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied on a component of the path prefix.

[ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist or the *path* argument points to an empty string.

[EPERM] The effective user ID does not match the owner of the file and the calling process does not have the appropriate privileges.

[EROFS] The named file resides on a read-only file system.

5.6.4.5 References

chown() §5.6.5, *mkdir*() §5.4.1, *mkfifo*() §5.4.2, *stat*() §5.6.2, <sys/stat.h> §5.6.1.

5.6.5 Change Owner and Group of a File

Function: *chown*()

5.6.5.1 Synopsis

```
#include <sys/types.h>

int chown (path, owner, group)
char *path;
uid_t owner, group;
```

5.6.5.2 Description

The *path* argument points to a pathname naming a file. The user ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

Only processes with an effective user ID equal to the user ID of the file or with appropriate privileges may change the ownership of a file. If {_POSIX_CHOWN_RESTRICTED} is in effect for *path*, this operation is restricted to processes with appropriate privileges. If {_POSIX_CHOWN_SUP_GRP} is in effect for *path*, the implementation limits a process with an effective user ID equal to the user ID of the file, but without appropriate privileges, to changing the group ID of a file only to the effective group ID of the process or to one of the supplementary group IDs.

The set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits of the file mode shall be cleared upon successful return from *chown*(), unless the the call is made by a process with appropriate privilege, in which case it is implementation defined whether those bits are altered. If the *chown*() function is successfully invoked on a file that is not a regular file, these bits may be cleared. These bits are defined in <sys/stat.h> §5.6.1.

Upon successful completion, the *chown*() function shall mark for update the *st_ctime* field of the file.

5.6.5.3 Returns

Upon successful completion, a value of zero shall be returned. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error. If −1 is returned, no change shall be made in the owner and group of the file.

5.6.5.4 Errors

If any of the following conditions occur, the *chown*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied on a component of the path prefix.

[ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist or the *path* argument points to an empty string.

[EPERM] The effective user ID does not match the owner of the file or the calling process does not have appropriate privileges.

[EROFS] The named file resides on a read-only file system.

For each of the following conditions, if the condition is detected, the *chmod*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The owner or group ID supplied is outside the range of zero to {UID_MAX}, inclusive.

5.6.5.5 References

chmod() §5.6.4, <sys/stat.h> §5.6.1.

5.6.6 Set File Access and Modification Times

Function: *utime*()

5.6.6.1 Synopsis

```
#include <sys/types.h>
#include <utime.h>

int utime (path, times)
char *path;
struct utimbuf *times;
```

5.6.6.2 Description

The argument *path* points to a pathname naming a file. The *utime*() function sets the access and modification times of the named file.

If the *times* argument is NULL, the access and modification times of the file are set to the current time. The effective user ID of the process must match the owner of the file, or the process must have write permission or appropriate privilege, to use the *utime*() function in this manner..

If *times* is not NULL, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. If {_POSIX_UTIME_OWNER} is in effect for *path*, the owner of the file shall be permitted to use the *utime*() function in this way, otherwise such use shall be restricted to processes with appropriate privileges.

The *utimbuf* structure is defined by the header <utime.h>, and includes the following members:

Member Type	Member Name	Description
time_t	actime	Access time
time_t	modtime	Modification time

The times in the *utimbuf* structure are measured in seconds since the Epoch (see Epoch §2.3).

Upon successful completion, the *utime*() function shall mark for update the *st_ctime* field of the file.

5.6.6.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned, *errno* is set to indicate the error, and the file times shall not be affected.

5.6.6.4 Errors

If any of the following conditions occur, the *utime*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied by a component of the path prefix; or the *times* argument is NULL and the effective user ID of the process does not match the owner of the file and write access is denied.

[ENAMETOOLONG]
The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOENT] The named file does not exist or the *path* argument points to an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The *times* argument is not NULL and the calling process's effective user ID has write access but does not match the owner of the file (if {_POSIX_UTIME_OWNER} is in effect) and the calling process does not have the appropriate privileges.

[EROFS] The file resides on a read-only file system.

5.6.6.5 References

<sys/stat.h> §5.6.1.

5.7 Configurable Pathname Variables

5.7.1 Get Configurable Pathname Variables

Functions: *pathconf*(), *fpathconf*()

5.7.1.1 Synopsis

```
long pathconf (path, name)
char *path;
int name;

long fpathconf (fildes, name)
int fildes, name;
```

5.7.1.2 Description

The *pathconf*() and *fpathconf*() functions provide a method for the application to determine the current value of a configurable limit or option (*variable*) that is associated with a file or directory.

For *pathconf*(), the *path* argument points to the pathname of a file or directory. For *fpathconf*(), the *fildes* argument is an open file descriptor.

The *name* argument represents the variable to be queried relative to that file or directory. The following table lists the pathname variables from <limits.h> §2.9 or <unistd.h> §2.10 that can be gotten by *pathconf*() or *fpathconf*(), and the symbolic constants, defined in <unistd.h>, that are the corresponding values used for *name*:

Variable	*name* Value
FCHR_MAX	_PC_FCHR_MAX
LINK_MAX	_PC_LINK_MAX
MAX_CANON	_PC_MAX_CANON
MAX_INPUT	_PC_MAX_INPUT
NAME_MAX	_PC_NAME_MAX
PATH_MAX	_PC_PATH_MAX
PIPE_BUF	_PC_PIPE_BUF
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED
_POSIX_CHOWN_SUP_GRP	_PC_CHOWN_SUP_GRP
_POSIX_DIR_DOTS	_PC_DIR_DOTS
_POSIX_GROUP_PARENT	_PC_GROUP_PARENT
_POSIX_LINK_DIR	_PC_LINK_DIR
_POSIX_NO_TRUNC	_PC_NO_TRUNC
_POSIX_UTIME_OWNER	_PC_UTIME_OWNER
_POSIX_V_DISABLE	_PC_V_DISABLE

5.7.1.3 Returns

If the variable corresponding to *name* is not defined on the system, or if *name* is an invalid value, or if the implementation does not support the association of *name* with the file specified by *path*, or if the process did not have the appropriate privileges to query the file specified by *path*, or *path* does not exist, the *pathconf*() function returns −1.

If the variable corresponding to *name* is not defined on the system, or if *name* is an invalid value, or if the implementation does not support the association of *name* with the file specified by *filedes*, the *fpathconf*() function returns −1.

Otherwise, the *pathconf*() or *fpathconf*() functions return the current variable value for the file or directory. The value returned shall not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's **<limits.h>** §2.9 or **<unistd.h>** §2.10.

6. Input and Output Primitives

The functions in this chapter deal with input and output from files and pipes. Functions are also specified which deal with the coordination and management of file descriptors and I/O activity.

6.1 Pipes

6.1.1 Create an Inter-Process Channel

Function: *pipe*()

6.1.1.1 Synopsis

```
int pipe (fildes)
int fildes[2];
```

6.1.1.2 Description

The *pipe*() function shall create a pipe and place two file descriptors, one each into the arguments *fildes*[0] and *fildes*[1], that refer to the open file descriptions for the read and write end of the pipe. Their integer values shall be the two lowest available at the time of the *pipe*() function call. The O_NONBLOCK flag shall be clear on both file descriptors. (The *fcntl*() function can be used to set the O_NONBLOCK flag.)

Data can be written to file descriptor *fildes*[1] and read from file descriptor *fildes*[0]. A read on file descriptor *fildes*[0] shall access the data written to file descriptor *fildes*[1] on a first-in-first-out basis.

An attempt to write on *fildes*[0] or to read on *fildes*[1] shall fail.

Upon successful completion, the *pipe*() function shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the pipe.

6.1.1.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

6.1.1.4 Errors

If any of the following conditions occur, the *pipe*() function shall return −1 and set *errno* to the corresponding value:

[EMFILE] More than {OPEN_MAX} minus two file descriptors are already in use by this process.

[ENFILE] The number of simultaneously open files in the system would exceed a system-imposed limit.

6.1.1.5 References

fcntl() §6.5.2, *open*() §5.3.1, *read*() §6.4.1, *write*() §6.4.2.

6.2 File Descriptor Manipulation

6.2.1 Duplicate an Open File Descriptor

Functions: *dup*(), *dup2*()

6.2.1.1 Synopsis

```
int dup (fildes)
int fildes;

int dup2 (fildes, fildes2)
int fildes, fildes2;
```

6.2.1.2 Description

The *dup*() and *dup2*() functions provide an alternate interface to the service provided by the *fcntl*() function using the F_DUPFD command. The call

```
fid = dup (fildes);
```

shall be equivalent to

```
fid = fcntl (fildes, F_DUPFD, 0);
```

The call

```
fid = dup2 (fildes, fildes2);
```

shall be equivalent to

```
close (fildes2);
fid = fcntl (fildes, F_DUPFD, fildes2);
```

except for the following:

If *fildes2* is not a valid file descriptor, the *dup2*() function shall return [EBADF].

If *fildes* is a valid file descriptor and is equal to *fildes2*, the *dup2*() function shall return *fildes2* without closing it.

6.2.1.3 Returns

Upon successful completion, the function shall return a file descriptor. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

6.2.1.4 Errors

If any of the following conditions occur, the *dup*() and *dup2*() functions shall return −1 and set *errno* to the corresponding value:

[EBADF] The argument *fildes* is not a valid file descriptor or *fildes2* is out of range.

[EMFILE] The number of file descriptors would exceed {OPEN_MAX}.

6.2.1.5 References

close() §6.3.1, *creat*() §5.3.2, *exec* §3.1.2, *fcntl*() §6.5.2, *open*() §5.3.1, *pipe*() §6.1.1.

6.3 File Descriptor Deassignment

6.3.1 Close a File

Function: *close*()

6.3.1.1 Synopsis

```
int close (fildes)
int fildes;
```

6.3.1.2 Description

The *fildes* argument is a file descriptor. The *close*() function shall deallocate (i.e., make available for return by subsequent *open*()'s, etc., executed by the process) the file descriptor indicated by *fildes*. All outstanding record locks owned by the process on the file descriptor indicated by *fildes* shall be removed (that is, unlocked).

If the *close*() function is interrupted by a signal that is to be caught, it shall return −1 with *errno* set to [EINTR] and the state of *fildes* is implementation defined. When all file descriptors associated with a pipe or FIFO special file have been closed, any data remaining in the pipe or FIFO shall be discarded.

6.3.1.3 Returns

Upon successful completion, a value of zero shall be returned. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

6.3.1.4 Errors

If any of the following conditions occur, the *close*() function shall return −1 and set *errno* to the corresponding value:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINTR] The *close* function was terminated prematurely by a signal.

6.3.1.5 References

creat() §5.3.2, *dup*() §6.2.1, *exec* §3.1.2, *fcntl*() §6.5.2, *fork*() §3.1.1, *open*() §5.3.1, *pipe*() §6.1.1.

6.4 Input and Output

6.4.1 Read from a File

Function: *read*()

6.4.1.1 Synopsis

```
int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

6.4.1.2 Description

The *fildes* argument is an open file descriptor.

The *read*() function shall attempt to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On a regular file or other file capable of seeking, *read*() shall start at a position in the file given by the file offset associated with *fildes*. Before successful return from *read*(), the file offset shall be incremented by the number of bytes actually read.

On a file not capable of seeking, the *read*() shall start from the current position. The value of a file offset associated with such a file is undefined.

Upon successful completion, the *read*() function shall return the number of bytes actually read and placed in the buffer. This number shall never be greater than *nbyte*. The value returned may be less than *nbyte* if the number of bytes left in the file is less than *nbyte*, if the *read*() request was interrupted by a signal, or if the file is a pipe (or FIFO) or special file and has fewer than *nbyte* bytes immediately available for reading. For example, a *read*() from a file associated with a terminal may return one typed line of data.

If a *read*() is interrupted by a signal before it reads any data, it shall return −1 with *errno* set to [EINTR].

If a *read*() is interrupted by a signal after it has successfully read some data, either it shall return −1 with *errno* set to [EINTR], or it shall return the number of bytes read. A *read*() from a pipe or FIFO shall never return with *errno* set to [EINTR] if it has transferred any data.

If an end-of-file has been reached, zero shall be returned. The result of subsequent *read*() requests on *fildes* is implementation defined.

The value of *nbyte* shall not be greater than {INT_MAX}; otherwise, the result is implementation defined.

When attempting to read from an empty pipe (or FIFO):

> If no process has the pipe open for writing, *read*() shall return zero to indicate end-of-file.
>
> If some process has the pipe open for writing and O_NONBLOCK is set, *read*() shall return a −1 and set *errno* [EAGAIN].
>
> If some process has the pipe open for writing and O_NONBLOCK is clear, *read*() shall block until some data is written or the pipe is closed by all processes that had opened the pipe for writing.

When attempting to read a file (other than a pipe or FIFO) that supports nonblocking reads and has no data currently available:

> If O_NONBLOCK is set, *read*() shall return a −1 and set *errno* to [EAGAIN].
>
> If O_NONBLOCK is clear, *read*() shall block until some data becomes available.
>
> The use of the O_NONBLOCK flag has no effect if there is some data available.

For any portion of a regular file, prior to the end-of-file, that has not been written, *read*() shall return bytes with value zero.

Upon successful completion, the *read*() function shall mark for update the *st_atime* field of the file.

6.4.1.3 Returns

Upon successful completion, *read*() shall return an integer indicating the number of bytes actually read. Otherwise, *read*() shall return a value of −1 and set *errno* to indicate the error, and the content of the buffer pointed to by *buf* is indeterminate.

6.4.1.4 Errors

If any of the following conditions occur, the *read*() function shall return −1 and set *errno* to the corresponding value:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the read operation.

[EBADF] The *fildes* argument is not a valid file descriptor open for reading.

[EINTR] The read operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.

6.4.1.5 References

creat() §5.3.2, *dup*() §6.2.1, *fcntl*() §6.5.2, *lseek*() §6.5.3, *open*() §5.3.1, *pipe*() §6.1.1, *sigaction*() §3.3.4.

6.4.2 Write to a File

Function: *write*()

6.4.2.1 Synopsis

```
int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

6.4.2.2 Description

The *fildes* argument is an open file descriptor.

The *write*() function shall attempt to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On a regular file or other file capable of seeking, the actual writing of data shall proceed from the position in the file indicated by the file offset associated with *fildes*. Before successful return from *write*(), the file offset shall be incremented by the number of bytes actually written.

On a file not capable of seeking, the *write*() shall start from the current position. The value of a file offset associated with such a file is undefined.

If the O_APPEND flag of the file status flags is set, the file offset shall be set to the end of the file prior to each write.

If a *write*() requests that more bytes be written than there is room for (for example, the physical end of a medium), only as many bytes as there is room for shall be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes would return 20. The next write of a non-zero number of bytes would give a failure return (except as noted below).

Upon successful completion, the *write*() function shall return the number of bytes actually written to the file associated with *fildes*. This number shall never be greater than *nbyte*.

If a *write*() is interrupted by a signal before it writes any data, it shall return −1 with *errno* set to [EINTR].

If *write*() is interrupted by a signal after it successfully writes some data, either it shall return −1 with *errno* set to [EINTR], or it shall return the number of bytes written. A *write*() to a pipe or FIFO shall never return with *errno* set to [EINTR] if it has transferred any data and *nbyte* is less than or equal to {PIPE_BUF}.

The value of *nbyte* shall not be greater than {INT_MAX}; otherwise, the result is implementation defined.

Write requests to a pipe (or FIFO) shall be handled the same as a regular file with the following exceptions:

> There is no file offset associated with a pipe, hence each write request shall append to the end of the pipe.
>
> Write requests of {PIPE_BUF} bytes or less shall not be interleaved with data from other processes doing writes on the same pipe. Writes of greater than {PIPE_BUF} bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the O_NONBLOCK flag of the file status flags is set.
>
> If the O_NONBLOCK flag is clear, a write request may cause the process to block, but on normal completion it shall return *nbyte*.
>
> If the O_NONBLOCK flag is set, *write*() requests shall be handled differently, in the following ways: the *write*() function shall not block the process; write requests for {PIPE_BUF} or fewer bytes shall either succeed completely and return *nbyte*, or return −1 and set *errno* to [EAGAIN].

When attempting to write to a file descriptor (other than a pipe or FIFO) that supports nonblocking writes and cannot accept the data immediately:

> If the O_NONBLOCK flag is clear, *write*() shall block until the data can be accepted.
>
> If the O_NONBLOCK flag is set, *write*() shall not block the process. If some data can be written without blocking the process, *write*() shall write what it can and return the number of bytes written. Otherwise, it shall return −1 and *errno* shall be set to [EAGAIN].

Upon successful completion, the *write*() function shall mark for update the *st_ctime* and *st_mtime* fields of the file.

6.4.2.3 Returns

Upon successful completion, *write*() shall return an integer indicating the number of bytes actually written. Otherwise, it shall return a value of −1 and set *errno* to indicate the error.

6.4.2.4 Errors

If any of the following conditions occur, the *write*() function shall return −1 and set *errno* to the corresponding value:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the write operation.

[EBADF] The *fildes* argument is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write a file that exceeds an implementation defined maximum file size.

[EINTR] The write operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfers for this file.

[ENOSPC] There is no free space remaining on the device containing the file.

[EPIPE] An attempt is made to write to a pipe (or FIFO) that is not open for reading by any process. A SIGPIPE signal shall also be sent to the process.

6.4.2.5 References

creat() §5.3.2, *dup*() §6.2.1, *fcntl*() §6.5.2, *lseek*() §6.5.3, *open*() §5.3.1, *pipe*() §6.1.1, *sigaction*() §3.3.4.

6.5 Control Operations on Files

6.5.1 Data Definitions for File Control Operations

6.5.1.1 Synopsis

```
#include <fcntl.h>
```

6.5.1.2 Description

The header <fcntl.h> §6.5.1 defines the following *requests* and *arguments* for the *fcntl*() and *open*() functions.

6.5.1.2.1 *cmd* values for *fcntl*()

Constant	Description
F_DUPFD	Duplicate file descriptor
F_GETFD	Get file descriptor flags
F_GETLK	Get record locking information
F_SETFD	Set file descriptor flags
F_GETFL	Get file status flags
F_SETFL	Set file status flags
F_SETLK	Set record locking information
F_SETLKW	Set record locking information; wait if blocked

6.5.1.2.2 File descriptor flags used for *fcntl*()

Constant	Description
FD_CLOEXEC	Close the file descriptor upon execution of an *exec* function

6.5.1.2.3 *l_type* values for record locking with *fcntl*()

Constant	Description
F_RDLCK	Shared or read lock
F_UNLCK	Unlock
F_WRLCK	Exclusive or write lock

6.5.1.2.4 *oflag* values for *open*()

Constant	Description
O_CREAT	Create file if it doesn't exist
O_EXCL	Exclusive use flag
O_TRUNC	Truncate flag

6.5.1.2.5 File status flags used for *open*() and *fcntl*()

Constant	Description
O_APPEND	Set append mode
O_NONBLOCK	No delay

6.5.1.2.6 File access modes used for *open*() and *fcntl*()

Constant	Description
O_RDONLY	Open for reading only
O_RDWR	Open for reading and writing
O_WRONLY	Open for writing only

6.5.1.2.7 Mask for use with file access modes

Constant	Description
O_ACCMODE	Mask for file access modes

6.5.1.3 References

fcntl() §6.5.2, *open*() §5.3.1.

6.5.2 File Control

Function: *fcntl*()

6.5.2.1 Synopsis

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl (fildes, cmd, ...)
int fildes, cmd;
```

6.5.2.2 Description

The function *fcntl*() provides for control over open files. The argument *fildes* is a file descriptor.

The available values for *cmd* are defined in the header <fcntl.h> §6.5.1, which shall include:

F_DUPFD — Return a new file descriptor which is the lowest numbered available (i.e., not already open) file descriptor greater than or equal to the third argument, *arg*, taken as an integer of type *int*. The new file descriptor refers to the same open file description as the original file descriptor, and shares any locks.

The FD_CLOEXEC flag associated with the new file descriptor is cleared to keep the file open across calls to the *exec* family of functions.

F_GETFD — Get the file descriptor flags defined in Table 6.5.1.2.2 that are associated with the file descriptor *fildes*. If the FD_CLOEXEC bit in the third argument, taken as type *int*, is zero the file shall remain open across *exec* functions; otherwise the file shall be closed upon successful execution of the *exec* function. File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file.

F_SETFD — Set the file descriptor flags defined in Table 6.5.1.2.2, that are associated with *fildes*, to the third argument, *arg*, taken as type *int*. This is zero or FD_CLOEXEC, as described for F_GETFD.

F_GETFL — Get the file status flags, defined in Table 6.5.1.2.5, and file access modes for the open file description associated with *fildes*. The file access modes defined in Table 6.5.1.2.6 can be extracted from the return value using the mask O_ACCMODE, which is defined in <fcntl.h> §6.5.1. File status flags and file access modes are associated with the open file description and do not affect other file descriptors that refer to the same file with different open file descriptions.

F_SETFL — Set the file status flags, defined in Table 6.5.1.2.5, for the open file description associated with *fildes* from the corresponding bits in the third argument, *arg*, taken as type *int*. The file access mode shall not be changed. If any other bits are set in *arg*, the result is implementation defined.

The following commands are available for record locking. Record locking shall be supported for regular files, and may be supported for other files.

F_GETLK
: Get the first lock which blocks the lock description pointed to by the third argument, *arg*, taken as a pointer to type *struct flock* (see below). The information retrieved overwrites the information passed to *fcntl*() in the structure *flock*. If no lock is found that would prevent this lock from being created, then the structure shall be left unchanged except for the lock type which shall be set to F_UNLCK.

F_SETLK
: Set or clear a file segment lock according to the lock description pointed to by the third argument, *arg*, taken as a pointer to type *struct flock* (see below). F_SETLK is used to establish shared (or read) locks (F_RDLCK) or exclusive (or write) locks, (F_WRLCK), as well as remove either type of lock (F_UNLCK). F_RDLCK, F_WRLCK, and F_UNLCK are defined by the <fcntl.h> §6.5.1 header. If a shared or exclusive lock cannot be set, *fcntl*() shall return immediately.

F_SETLKW
: This command is the same as F_SETLK except that if a shared or exclusive lock is blocked by other locks, the process shall wait until the request can be satisfied. If a signal that is to be caught is received while *fcntl*() is waiting for a region, the *fcntl*() shall be interrupted. Upon return from the process's signal handler, *fcntl*() shall return −1 with *errno* set to [EINTR], and the lock operation shall not be done.

The structure *flock*, defined by the <fcntl.h> §6.5.1 header, describes a lock. It describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), and process-ID (*l_pid*):

Member Type	Member Name	Description
short	l_type	F_RDLCK, F_WRLCK, F_UNLCK
short	l_whence	flag for starting offset
off_t	l_start	relative offset in bytes
off_t	l_len	size; if 0 then until EOF
int	l_pid	process ID of the process holding the lock, returned with F_GETLK

When a shared lock has been set on a segment of a file, other processes shall be able to set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock shall fail if the file descriptor was not opened with read access.

An exclusive lock shall prevent any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock shall fail if the file descriptor was not opened with write access.

The value of *l_whence* is SEEK_SET, SEEK_CUR, or SEEK_END to indicate that the relative offset, *l_start* bytes, will be measured from the start of the file, current position, or end of the file, respectively. The value of *l_len* is the number of consecutive bytes to be locked. If *l_len* is negative, the result is implementation defined. The *l_pid* field is only used with F_GETLK to return the process ID of the process holding a blocking lock.

Locks may start and extend beyond the current end of a file, but shall not start or extend before the beginning of the file. A lock shall be set to extend to the end of file if *l_len* is set to zero. If the *flock struct* has *l_whence* and *l_start* that point to the beginning of the file, and *l_len* of zero, the entire file shall be locked.

The calling process shall have only one type of lock set for each byte in the file. Before successful return from a F_SETLK or F_SETLKW request, the previous lock type for each byte in the specified region shall be replaced by the new lock type. All locks associated with a file for a given process shall be removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process created using the *fork*() function.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock another process's locked region. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, the *fcntl*() function shall fail with an [EDEADLK] error.

6.5.2.3 Returns

Upon successful completion, the value returned shall depend on *cmd* as follows:

Request	Return Value
F_DUPFD	A new file descriptor.
F_GETFD	Value of the flags defined in Table 6.5.1.2.2, but the return value shall not be negative.
F_SETFD	Value other than −1.
F_GETFL	Value of file status flags and access modes, but the return value shall not be negative.
F_SETFL	Value other than −1.
F_GETLK	Value other than −1.
F_SETLK	Value other than −1.
F_SETLKW	Value other than −1.

Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

6.5.2.4 Errors

If any of the following conditions occur, the *fcntl*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] The argument *cmd* is F_SETLK, the type of lock (*l_type*) is a shared lock (F_RDLCK) or exclusive lock (F_WRLCK), and the segment of a file to be locked is already exclusive-locked by another process, or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

[EBADF] The *fildes* argument is not a valid file descriptor.

The argument *cmd* is F_SETLK or F_SETLKW, the type of lock (*l_type*) is a shared lock (F_RDLCK), and *fildes* is not a valid file descriptor open for reading.

The argument *cmd* is F_SETLK or F_SETLKW, the type of lock (*l_type*) is an exclusive lock (F_WRLCK), and *fildes* is not a valid file descriptor open for writing.

[EINTR] The argument *cmd* is F_SETLKW and the function was interrupted by a signal.

[EINVAL] The argument *cmd* is F_DUPFD and the third argument is negative or greater than or equal to {OPEN_MAX}.

The argument *cmd* is F_GETLK, F_SETLK, or F_SETLKW and the data *arg* points to is not valid, or *fildes* refers to a file that does not support locking.

[EMFILE] The argument *cmd* is F_DUPFD and {OPEN_MAX} file descriptors are currently in use by this process.

[ENOLCK] The argument *cmd* is F_SETLK or F_SETLKW and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.

For each of the following conditions, if the condition is detected, the *fcntl*() function shall return −1 and set *errno* to the corresponding value:

[EDEADLK] The argument *cmd* is F_SETLKW and a deadlock condition was detected.

6.5.2.5 References

close() §6.3.1, *exec* §3.1.2, *open*() §5.3.1, <fcntl.h> §6.5.1, *sigaction*() §3.3.4.

6.5.3 Reposition Read/Write File Offset

Function: *lseek*()

6.5.3.1 Synopsis

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek (fildes, offset, whence)
int fildes, whence;
off_t offset;
```

6.5.3.2 Description

The *fildes* argument is an open file descriptor. The *lseek*() function shall set the file offset for the open file description associated with *fildes* as follows:

If *whence* is SEEK_SET, the offset is set to *offset* bytes.

If *whence* is SEEK_CUR, the offset is set to its current value plus *offset* bytes.

If *whence* is SEEK_END, the offset is set to the size of the file plus *offset* bytes.

The symbolic constants SEEK_SET, SEEK_CUR, SEEK_END are defined in the header <unistd.h> §2.10.

Some devices are incapable of seeking. The value of the file offset associated with such a device is undefined.

The *lseek*() function shall allow the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads of data in the gap shall return bytes with the value zero until data is actually written into the gap.

The *lseek*() function shall not, by itself, extend the size of a file.

6.5.3.3 Returns

Upon successful completion, the function shall return the resulting offset location as measured in bytes from the beginning of the file. Otherwise, it shall return a value of (off_t) −1, shall set *errno* to indicate the error, and the file offset shall remain unchanged.

6.5.3.4 Errors

If any of the following conditions occur, the *lseek*() function shall return −1 and set *errno* to the corresponding value:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] The *whence* argument is not a proper value, or the resulting file offset would be invalid.

[ESPIPE] The *fildes* argument is associated with a pipe or FIFO.

6.5.3.5 References

creat() §5.3.2, *dup*() §6.2.1, *fcntl*() §6.5.2, *open*() §5.3.1, *read*() §6.4.1, *sigaction*() §3.3.4, *write*() §6.4.2, **<unistd.h>** §2.10.

7. Device- and Class-Specific Functions

7.1 General Terminal Interface

7.1.1 Interface Characteristics

7.1.1.1 Description

This section describes a general terminal interface that shall be provided to control asynchronous communications ports. It is implementation defined whether this interface supports network connections and/or synchronous ports.

7.1.1.2 Opening a Terminal Device File

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, user programs seldom open these files; they are opened by special programs and become a user's standard input, output, and error files.

As described in *open*() §5.3.1, opening a terminal device file with the O_NONBLOCK flag clear shall cause the process to block until the connection is established. If the O_NONBLOCK flag is set, the *open*() function shall return a file descriptor without waiting for a connection to be established.

7.1.1.3 Process Groups

A terminal may have a distinguished process group associated with it. This distinguished process group plays a special role in handling signal-generating input characters, as discussed below in Special Characters §7.1.1.10.

If the implementation supports the Job Control Option (if {_POSIX_JOB_CONTROL} is defined; see Symbolic Constants §2.10), command interpreter processes* supporting job control can allocate the terminal to different *jobs*, or process groups, by placing related processes in a single process group and associating this process group with the terminal. A terminal's associated process group may be set or examined by a process, assuming the

* The P1003.2 Working Group is working on a definition and description of command interpreters. See Shell and Utilities §A.2.2.

permission requirements in this section are met; see *tcgetpgrp*() §7.2.3 and *tcsetpgrp*() §7.2.4. The terminal interface aids in this allocation by restricting access to the terminal by processes that are not in the current process group; see Job Access Control §7.1.1.5.

7.1.1.4 The Controlling Terminal

A terminal may belong to a process as its *controlling terminal*. If a process that is a "session process group leader," and that does not have a controlling terminal, opens a terminal file not already associated with a process group, the terminal associated with that terminal file becomes the controlling terminal for that process, and the terminal's distinguished process group is set to the process group of that process.

The controlling terminal is inherited by a child process during a *fork*() function. A process relinquishes its controlling terminal when it changes its process group using a *setpgrp*() function.

When controlling process terminates, the distinguished process group of its controlling terminal is set to zero (indicating no distinguished process group). This allows the terminal to be acquired as a controlling terminal by a new session process group leader.

7.1.1.5 Job Access Control

If a process is in the distinguished process group of its controlling terminal, or the distinguished process group is zero (that is, if the process is a *foreground process*), then read operations shall be allowed as described below in Input Processing and Reading Characters §7.1.1.6. For those implementations that do not support the Job Control Option, a background process shall also be allowed to read from its controlling terminal. For those implementations that support the Job Control Option, if a process is not in the (non-zero) distinguished process group of its controlling terminal (that is, if the process is a *background process*), then any attempts to read from that terminal shall cause the process group to be sent a SIGTTIN signal unless the reading process is ignoring or blocking the SIGTTIN signal. If the process is ignoring or blocking the SIGTTIN signal, the process is instead returned an [EIO] error and no signal is sent to the process. The default action of the SIGTTIN signal is to stop the process to which it is sent. See Signal Names §3.3.1.

It is frequently undesirable for background processes to write to their controlling terminal. If TOSTOP (see Local Modes §7.1.2.6) is set, then attempts by a background process to write to its controlling terminal shall cause the process group to be sent a SIGTTOU signal, which, by default, will cause the members of the process group to stop. If TOSTOP is not set or the process is ignoring or blocking SIGTTOU signals, the process is allowed to write to the terminal and the SIGTTOU signal is not sent. Certain calls that set terminal parameters are treated in this same fashion, except that TOSTOP is ignored; however, the effect is identical to that of terminal writes when TOSTOP is set. See Control Functions §7.2.

7.1.1.6 Input Processing and Reading Characters

A terminal device associated with a terminal device file may operate in full-duplex mode, so that characters may arrive even while output is occurring. Each terminal device file has associated with it an *input queue*, into which incoming characters are stored by the system before being read by a process. The system may impose a limit, {MAX_INPUT}, on the number of bytes that may be stored in the input queue. The behavior of the system when this limit is exceeded is implementation defined.

Two general kinds of input processing are available, determined by whether the terminal device file is in canonical mode or non-canonical mode. These modes are described in the Canonical Mode Input Processing §7.1.1.7 and Non-Canonical Mode Input Processing §7.1.1.8. Additionally, input characters are processed according to the *c_iflag* (see Input Modes §7.1.2.3) and *c_lflag* (see Local Modes §7.1.2.6) fields. Such processing can include *echoing*, which in general means transmitting input characters immediately back to the terminal when they are received from the terminal. This is useful for terminals that can operate in full-duplex mode. The manner in which characters are provided to a process reading from a terminal device file is very dependent on whether the terminal file is in canonical or non-canonical mode.

Another dependency is whether the O_NONBLOCK flag is set by *open*() or *fcntl*(). If the O_NONBLOCK flag is clear, then the read request shall shall block until data is available or a signal has been received. If the O_NONBLOCK flag is set, then the read request shall complete, without blocking, in one of three ways:

1. If there is enough data available to satisfy the entire request, the read shall complete successfully, having read all the requested data, and return the number of bytes read.

2. If there is not enough data available to satisfy the entire request, the read shall complete successfully, having read as much data as possible, and return the number of bytes it was able to read.

3. If there is no data available, the read shall return a −1, with *errno* set to EAGAIN.

When data is available depends on whether the input processing mode is canonical or non-canonical. The following sections, Canonical Mode Input Processing §7.1.1.7 and Non-Canonical Mode Input Processing §7.1.1.8, describe each of these input processing modes.

7.1.1.7 Canonical Mode Input Processing

In canonical mode input processing, terminal input is processed in units of lines. A line is delimited by a new-line ('\n'). character, an end-of-file (EOF) character, or an end-of-line (EOL) character. See the Special Characters §7.1.1.10 for more information on EOF and EOL. This means that a read request shall not be satisfied until an entire line has been typed, or a signal has been received. Also, no matter how many characters are requested in the read call, at most one line shall be returned. It is not, however, necessary to read a whole line at once; any number of characters, even one, may be requested in a read without losing information.

If {MAX_CANON} is defined, it is a limit on the number of bytes in a line. The behavior of the system when this limit is exceeded is implementation defined.

Erase and kill processing occurs during input. The ERASE character erases the last character typed in the current input line.

ERASE shall not erase beyond the beginning of the current input line. The KILL character kills (deletes) the entire current input line, and optionally outputs a new-line character. All these characters operate on a keystroke basis, independently of any backspacing or tabbing that may have been done.

7.1.1.8 Non-Canonical Mode Input Processing

In non-canonical mode input processing, input characters are not assembled into lines, and erase and kill processing does not occur. The values of the special characters MIN and TIME are used to determine how to process the characters received. MIN and TIME are defined by the *c_cc* array of special control characters.

MIN represents the minimum number of characters that should be received when the read is satisfied (i.e., the characters are returned to the user). TIME is a timer of 0.1 second granularity that is used to time out bursty and short term data transmissions. If MIN is greater than {MAX_INPUT}, the response to the request is implementation defined. The four possible values for MIN and TIME and their interactions are described below.

7.1.1.8.1 Case A: MIN > 0, TIME > 0

In this case TIME serves as an intercharacter timer and is activated after the first character is received. Since it is an intercharacter timer, it is reset after a character is received. The interaction between MIN and TIME is as follows: as soon as one character is received, the intercharacter timer is started. If MIN characters are received before the intercharacter timer expires (remember that the timer is reset upon receipt of each character), the read is satisfied. If the timer expires before MIN characters are received, the characters received to that point are returned to the user. Note that if TIME expires at least one character shall be returned because the timer would not have been enabled unless a character was received. In this case (MIN > 0, TIME > 0) the read shall block until the MIN and TIME mechanisms are activated by the receipt of the first character.

7.1.1.8.2 Case B: MIN > 0, TIME = 0

In this case, since the value of TIME is zero, the timer plays no role and only MIN is significant. A pending read is not satisfied until MIN characters are received (i.e., the pending read shall block until MIN characters are received). A program that uses this case to read record-based terminal I/O may block indefinitely in the read operation.

7.1.1.8.3 Case C: MIN = 0, TIME > 0

In this case, since MIN = 0, TIME no longer represents an intercharacter timer. It now serves as a read timer that is activated as soon as the *read*() function is processed. A read is satisfied as soon as a single character is received or the read timer expires. Note that in this case if the timer expires, no character shall be returned. If the timer does not expire, the only way the read can be satisfied is if a character is received. In this case the read shall not block indefinitely waiting for a character; if no character is received within TIME*0.1 seconds after the read is initiated, the read shall return with zero characters.

7.1.1.8.4 Case D: MIN = 0, TIME = 0

The minimum of either the number of characters requested or the number of characters currently available shall be returned without waiting for more characters to be input.

7.1.1.8.5 Comparison of the Different Cases of MIN, TIME Interaction

Some points to note about MIN and TIME:

1. In the preceding explanations one may notice that 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 intercharacter 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 (e.g., file transfer programs) where a program would like to process at least MIN characters at a time. In case A, the intercharacter timer is activated by a user as a safety measure; while 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; while 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.

7.1.1.9 Writing Characters and Output Processing

When a process writes one or more characters to a terminal device file, they are processed according to the *c_oflag* (see Output Modes §7.1.2.4) The implementation may provide a buffering mechanism; as such, when a call to *write*() completes, all of the characters written have been scheduled for transmission to the device, but the transmission will not necessarily have completed.

7.1.1.10 Special Characters

Certain characters have special functions on input and/or output. These functions are summarized as follows:

INTR — Special character on input and is recognized if the ISIG flag is enabled. Generates a SIGINT signal which is sent to all processes in the distinguished process group associated with the terminal

QUIT — Special character on input and is recognized if the ISIG flag is enabled. Generates a SIGQUIT signal which is sent to all processes in the distinguished process group associated with the terminal.

ERASE — Special character on input and is recognized if the ICANON flag is set. Erases the preceding character. It shall not erase beyond the start of a line, as delimited by an NL, EOF, or EOL character.

KILL — Special character on input and is recognized if the ICANON flag is set. Deletes the entire line, as delimited by a NL, EOF, or EOL character.

EOF — Special character on input and is recognized if the ICANON flag is set. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOF is discarded. Thus, if there are no characters waiting (that is, the EOF occurred at the beginning of a line), zero characters shall be passed back, representing an end-of-file indication.

NL — Special character on input and is recognized if the ICANON flag is set. Is the line delimiter ('\n'). It cannot be changed.

EOL — Special character on input and is recognized if the ICANON flag is set. Is an additional line delimiter, like NL.

SUSP — Special character on input and is recognized if the ISIG flag is enabled (Job Control Option only). Generates a SIGTSTP signal which is sent to all processes in the distinguished process group associated with the terminal.

STOP — Special character on both input and output and is recognized if the IXON (input) or IXOFF (output) flag is set. (ASCII DC3) Can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read.

START — Special character on both input and output and is recognized if the IXON (input) or IXOFF (output) flag is set. (ASCII DC1) Can be used to resume output that has been suspended by a STOP character.

The START and STOP characters cannot be changed. The values for INTR, QUIT, ERASE, KILL, EOF, EOL, SUSP (Job Control Option only), shall be changeable to suit individual tastes.

If {_POSIX_V_DISABLE} is in effect for the terminal file, special character functions can be disabled individually.

If two or more special characters have the same value, the function performed when that character is received is undefined.

A special character is recognized not only by its value, but also by its context; e.g., an implementation may define multi-byte sequences that have a meaning different from the meaning of the bytes when considered individually. Implementations may also define additional single-byte functions.

7.1.1.11 Modem Disconnect

When a modem disconnect is detected by the terminal interface, a SIGHUP signal is sent to all processes in the distinguished process group associated with the terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If SIGHUP is ignored or caught, any subsequent read returns with an end-of-file indication until the device is closed. Thus programs that read a terminal file and test for end-of-file can terminate appropriately after a disconnect.

7.1.1.12 Closing a Terminal Device File

The last process to close a terminal device file shall cause any output to be sent to the device and any input to be discarded. If HUPCL is set in the control modes, and the communications port supports a disconnect function, the terminal device shall perform a disconnect.

7.1.2 Settable Parameters

7.1.2.1 Synopsis

```
#include <termios.h>
```

7.1.2.2 *termios* Structure

Routines that need to control certain terminal I/O characteristics shall do so by using the *termios* structure as defined in the header <termios.h>. The members of this structure include (but are not limited to):

Member Type	Array Size	Member Name	Description
unsigned long		c_iflag	input modes
unsigned long		c_oflag	output modes
unsigned long		c_cflag	control modes
unsigned long		c_lflag	local modes
unsigned char [ ]	NCCS	c_cc	control chars

The total size of the *termios* structure is implementation defined.

7.1.2.3 Input Modes

The *c_iflag* field describes the basic terminal input control:

Mask Name	Description
BRKINT	Signal interrupt on break.
ICRNL	Map CR to NL on input.
IGNBRK	Ignore break condition.
IGNCR	Ignore CR.
IGNPAR	Ignore characters with parity errors.
INLCR	Map NL to CR on input.
INPCK	Enable input parity check.
ISTRIP	Strip character.
IXOFF	Enable start/stop input control.
IXON	Enable start/stop output control.
PARMRK	Mark parity errors.

If IGNBRK is set, a break condition (a character framing error with data all zeroes) detected on input is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise if BRKINT is set, the break condition shall generate a single SIGINT signal and flush both the input and output queues. If neither IGNBRK or BRKINT is set, a break condition is read as a single '\0'.

If IGNPAR is set, a byte with a framing or parity error (other than break) is ignored.

If PARMRK is set, a byte with a framing and parity error (other than break) that is not ignored is given to the application as the three-character sequence `'\377'`, `'\0'`, *X*, where `'\377'`, `'\0'` is a two-character flag preceding each sequence and *X* is the data of the character received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid character of `'\377'` is given to the application as `'\377'`, `'\377'`. If PARMRK is not set, a framing or parity error (other than break) that is not ignored is given to the application as a single character `'\0'`.

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled, allowing output parity generation without input parity errors. Note that whether input parity checking is enabled or disabled is independent of whether parity detection is enabled or disabled. If parity detection is enabled but input parity checking is disabled, the hardware to which the terminal is connected shall recognize the parity bit, but the terminal special file shall not check whether this bit is set correctly or not.

If ISTRIP is set, valid input characters are first stripped to 7 bits, otherwise all 8 bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). Otherwise if ICRNL is set, a received CR character is translated into a NL character.

If IXON is set, start/stop output control is enabled. A received STOP character shall suspend output and a received START character shall restart output. When IXON is set, START and STOP characters are not read, but merely perform flow control functions. When IXON is not set, the START and STOP characters are read.

If IXOFF is set, start/stop input control is enabled. The system shall transmit STOP characters, which are intended to cause the terminal device to stop transmitting data, as needed to prevent the number of characters in the input queue from exceeding {MAX_INPUT}, and shall transmit START characters, which are intended to cause the terminal device to resume transmitting data, as soon as the device can continue transmitting data without risk of overflowing the input queue. The precise conditions under which STOP and START characters are transmitted are implementation defined.

The initial input control value after *open*() is implementation defined.

7.1.2.4 Output Modes

The *c_oflag* field specifies the terminal interface's treatment of output:

Mask Name	Description
OPOST	Perform output processing.

If OPOST is set, output characters are processed in an implementation defined fashion so that lines of text are modified to appear appropriately on the terminal device, otherwise characters are transmitted without change.

The initial output control value after *open*() is implementation defined.

7.1.2.5 Control Modes

The *c_cflag* field describes the hardware control of the terminal; not all values specified are required to be supported by the underlying hardware:

Mask Name	Description
CLOCAL	Ignore modem status lines.
CREAD	Enable receiver.
CSIZE	Character size:
CS5	5 bits
CS6	6 bits
CS7	7 bits
CS8	8 bits
CSTOPB	Send two stop bits, else one.
HUPCL	Hang up on last close.
PARENB	Parity enable.
PARODD	Odd parity, else even.

In addition, the input and output baud rates are also stored in the *c_cflag* field. The following values are supported:

Name	Description
B0	Hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud

The following interfaces are provided for getting and setting the values of the input and output baud rates:

```
int cfgetospeed (termios_p)
struct termios *termios_p;

int cfsetospeed (termios_p, speed)
struct termios *termios_p;
int speed;

int cfgetispeed (termios_p)
struct termios *termios_p;

int cfsetispeed (termios_p, speed)
struct termios *termios_p;
int speed;
```

The *termios_p* argument is a pointer to a *termios* structure.

cfgetospeed() returns the output baud rate stored in *c_cflag* pointed to by *termios_p*.

cfsetospeed() sets the output baud rate stored in the *c_cflag* pointed to by *termios_p* to *speed*. The zero baud rate, B0, is used to terminate the connection. If B0 is specified, the modem control lines shall no longer be asserted. Normally, this will disconnect the line.

cfgetispeed() returns the input baud rate stored in *c_cflag*.

cfsetispeed() sets the input baud rate stored in *c_cflag* to *speed*. If the input baud rate is set to zero, the input baud rate will be specified by the value of the output baud rate. For any particular hardware, unsupported baud rate changes are ignored. This refers both to changes to baud rates not supported by the hardware, and to changes setting the input and output baud rates to different values if the hardware does not support this.

The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used, otherwise one stop bit. For example, at 110 baud, two stop bits are normally used.

If CREAD is set, the receiver is enabled. Otherwise, no characters shall be received.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, PARODD specifies odd parity if set, otherwise even parity is used.

If HUPCL is set, the modem control lines for the port shall be lowered when the last process with the port open closes the port or the process terminates. The modem connection shall be broken. If HUPCL is not set, the control lines are not altered.

If CLOCAL is set, a connection does not depend on the state of the modem status lines. If CLOCAL is clear, the modem status lines shall be monitored.

Under normal circumstances, a call to the *open*() function shall wait for the modem connection to complete. However, if the O_NONBLOCK flag is set (see *open*() §5.3.1) or if CLOCAL has been set, the *open*() function shall return immediately without waiting for the connection. For those files on which the connection has not been established, or on which a modem disconnect has occurred, and for which CLOCAL is not set, both *read*() and *write*() shall return a zero character count. For *read*(), this is equivalent to an end-of-file condition.

If the object for which the control modes are set is not an asynchronous serial connection, some of the modes may be ignored; e.g., if an attempt is made to set the baud rate on a network connection to a terminal on another host, the baud rate may or may not be set on the connection between that terminal and the machine it is directly connected to.

The initial hardware control value after *open*() is implementation defined.

7.1.2.6 Local Modes

The *c_lflag* field of the argument structure is used to control various functions:

Mask Name	Description
ECHO	Enable echo.
ECHOE	Echo ERASE as an error-correcting backspace.
ECHOK	Echo KILL.
ECHONL	Echo '\n'.
ICANON	Canonical input (erase and kill processing).
ISIG	Enable signals.
NOFLSH	Disable flush after interrupt, quit, or suspend.
TOSTOP	Send SIGTTOU for background output.

If ECHO is set, input characters are echoed back to the terminal. If ECHO is not set, input characters are not echoed.

If ECHOE and ICANON are set, the ERASE character shall cause the terminal to erase the character from the display, if possible.

If ECHOK and ICANON are set, the KILL character shall either cause the terminal to erase the line from the display or shall echo the '\n' character after the KILL character.

If ECHONL and ICANON are set, the '\n' character shall be echoed even if ECHO is not set.

If ISIG is set, each input character is checked against the special control characters INTR, QUIT, and SUSP (Job Control Option only). If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus these special input functions are possible only if ISIG is set.

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL, as described in Canonical Mode Input Processing §7.1.1.7.

If ICANON is not set, read requests are satisfied directly from the input queue. A read shall not be satisfied until at least MIN characters have been received or the timeout value TIME expired between characters. The time value represents tenths of seconds. See the Non-Canonical Mode Input Processing §7.1.1.8 section for more details.

If NOFLSH is set, the normal flush of the input and output queues associated with the INTR, QUIT, and SUSP (Job Control Option only) characters shall not be done.

If TOSTOP (Job Control Option only) is set, the signal SIGTTOU is sent to the process group of a process that tries to write to its controlling terminal if it is not in the distinguished process group for that terminal. This signal, by default, stops the members of the process group. Otherwise, the output generated by that process is output to the current output stream. Processes that are blocking or ignoring SIGTTOU signals are excepted and allowed to produce output and the SIGTTOU signal is not sent.

The initial local control value after *open*() is implementation defined.

7.1.2.7 Special Control Characters

The special control characters values are defined by the array *c_cc*. The subscript name and description for each element in both canonical and non-canonical modes are as follows:

Canonical Subscript	Non-Canonical Subscript	Description
VEOF		EOF character
VEOL		EOL character
VERASE		ERASE character
VINTR	VINTR	INTR character
VKILL		KILL character
	VMIN	MIN value
VQUIT	VQUIT	QUIT character
VSUSP	VSUSP	SUSP character
	VTIME	TIME value

The subscript values shall be unique, except that the VMIN and VTIME subscripts may have the same values as the VEOF and VEOL subscripts, respectively.

The VSUSP index shall be defined only if the Job Control Option is supported.

The number of elements in the *c_cc* array, NCCS, is implementation defined.

The initial values of all control characters are implementation defined.

If {_POSIX_V_DISABLE} is in effect for the terminal file, and the value of one of the special control characters is {_POSIX_V_DISABLE}, that function shall be disabled. The {_POSIX_V_DISABLE} character is always read if received, and never causes a special character function.

7.2 General Terminal Interface Control Functions

The functions that are used to control the general terminal function are described in this section. If the implementation supports the Job Control Option, unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to perform the operation and the SIGTTOU signal is not sent.

In all the functions, *fildes* is an open file descriptor.

7.2.1 Get and Set State

Functions: *tcgetattr*(), *tcsetattr*()

7.2.1.1 Synopsis

```
#include <termios.h>

int tcgetattr (fildes, termios_p)
int fildes;
struct termios *termios_p;

int tcsetattr (fildes, optional_actions, termios_p)
int fildes;
int optional_actions;
struct termios *termios_p;
```

7.2.1.2 Description

The *tcgetattr*() function shall get the parameters associated with the object referred to by *fildes* and store them in the *termios* structure referenced by *termios_p*. This command is allowed from a background process; however, the information may be subsequently changed by a foreground process.

The *tcsetattr*() function shall set the parameters associated with the terminal from the *termios* structure referenced by *termios_p* as follows:

If *optional_actions* is TCSANOW, the change shall occur immediately.

If *optional_actions* is TCSADRAIN, the change shall occur after all output written to *fildes* has been transmitted. This function should be used when changing parameters that affect output.

If *optional_actions* is TCSADFLUSH, the change shall occur after all output written to the object referred to by *fildes* has been transmitted, and all input that has been received but not read shall be discarded before the change is made.

7.2.1.3 Returns

Upon successful completion, a value of zero is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

7.2.1.4 Errors

If any of the following conditions occur, the *tcgetattr*() function shall return −1 and set *errno* to the corresponding value:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] The device does not support the *tcgetattr*() function.

[ENOTTY] The file associated with *fildes* is not a terminal

If any of the following conditions occur, the *tcsetattr*() function shall return −1 and set *errno* to the corresponding value:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] The device does not support the *tcsetattr*() function, or the *optional_actions* argument is not a proper value.

[ENOTTY] The file associated with *fildes* is not a terminal

7.2.1.5 References

<termios.h> §7.1.2.

7.2.2 Line Control Functions

Functions: *tcsendbreak*(), *tcdrain*(), *tcflush*(), *tcflow*()

7.2.2.1 Synopsis

```
#include <termios.h>

int tcsendbreak (fildes, duration)
int fildes;
int duration;

int tcdrain (fildes)
int fildes;

int tcflush (fildes, queue_selector)
int fildes;
int queue_selector;

int tcflow (fildes, action)
int fildes;
int action;
```

7.2.2.2 Description

The *tcsendbreak*() function shall send a "break"; that is, a continuous stream of zero-valued bits for a specific duration. If *duration* is zero, it shall send zero-valued bits for 0.25 seconds. If *duration* is not zero, it shall send zero-valued bits for an implementation defined period of time.

The *tcdrain*() function shall wait until all output written to the object referred to by *fildes* has been transmitted.

The *tcflush*() function shall discard data written to the object referred to by *fildes* but not transmitted, or data received but not read, depending on the value of *queue_selector*:

If *queue_selector* is TCIFLUSH, it shall flush data received but not read.

If *queue_selector* is TCOFLUSH, it shall flush data written but not transmitted.

If *queue_selector* is TCIOFLUSH, it shall flush both data received but not read, and data written but not transmitted.

The *tcflow*() function shall suspend transmission or reception of data on the object referred to by *fildes*, depending on the value of *action*:

If *action* is TCOOFF, it shall suspend output.

If *action* is TCOON, it shall restart suspended output.

If *action* is TCIOFF, it shall suspend input.

If *action* is TCION, it shall restart suspended input.

The default on open of a terminal file is that neither its input nor its output are suspended.

7.2.2.3 Returns

Upon successful completion, a value of zero is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

7.2.2.4 Errors

If any of the following conditions occur, the *tcsendbreak*() function shall return −1 and set *errno* to the corresponding value:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] The device does not support the *tcsendbreak*() function.

[ENOTTY] The file associated with *fildes* is not a terminal

If any of the following conditions occur, the *tcdrain*() function shall return −1 and set *errno* to the corresponding value:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINTR] A signal interrupted the *tcdrain*() function.

[EINVAL] The device does not support the *tcdrain*() function.

[ENOTTY] The file associated with *fildes* is not a terminal

If any of the following conditions occur, the *tcflush*() function shall return −1 and set *errno* to the corresponding value:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] The device does not support the *tcflush*() function, or the *queue_selector* argument is not a proper value.

[ENOTTY] The file associated with *fildes* is not a terminal

If any of the following conditions occur, the *tcflow*() function shall return −1 and set *errno* to the corresponding value:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] The device does not support the *tcflow*() function, or the *action* argument is not a proper value.

[ENOTTY] The file associated with *fildes* is not a terminal

7.2.2.5 References

<termios.h> §7.1.2.

7.2.3 Get Distinguished Process Group ID

Function: *tcgetpgrp*()

7.2.3.1 Synopsis

```
#include <termios.h>

int tcgetpgrp (fildes)
int fildes;
```

7.2.3.2 Description

The *tcgetpgrp*() function shall be provided if the implementation supports the Job Control Option.

The *tcgetpgrp*() function shall return the value of the process group ID of the distinguished process group associated with the terminal.

The *tcgetpgrp*() function is allowed from a background process; however, the information may be subsequently changed by a foreground process.

7.2.3.3 Returns

Upon successful completion, *tcgetpgrp*() returns the process group ID of the distinguished process group associated with the terminal. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

7.2.3.4 Errors

If any of the following conditions occur, the *tcgetpgrp*() function shall return −1 and set *errno* to the corresponding value:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] This function is not allowed for the device associated with the *fildes* argument.

[ENOTTY] The calling process does not have a controlling terminal or the file is not the controlling terminal.

7.2.3.5 References

setpgrp() §4.3.2, *jcsetpgrp*() §4.3.3, *tcsetpgrp*() §7.2.4.

7.2.4 Set Distinguished Process Group ID

Function: *tcsetpgrp*()

7.2.4.1 Synopsis

```
#include <termios.h>

int tcsetpgrp (fildes, pgrp_id)
int fildes;
int pgrp_id;
```

7.2.4.2 Description

The *tcsetpgrp*() function shall be provided if the implementation supports the Job Control Option.

If the process has a controlling terminal, the *tcsetpgrp*() function shall set the distinguished process group ID associated with the terminal to *pgrp_id*. The file associated with *fildes* must be the controlling terminal of the calling process. There must be at least one process in *pgrp_id* that has the same controlling terminal as the calling process.

7.2.4.3 Returns

Upon successful completion, *tcsetpgrp*() returns a value of zero. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

7.2.4.4 Errors

If any of the following conditions occur, the *tcsetpgrp*() function shall return −1 and set *errno* to the corresponding value:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] This function is not allowed for the device associated with the *fildes* argument or the value of the *pgrp_id* argument is less than or equal to zero, or exceeds {PID_MAX}.

[ENOTTY] The calling process does not have a controlling terminal or the file is not the controlling terminal.

[EPERM] The value of the *pgrp_id* argument is greater than zero and less than or equal to {PID_MAX}, and there is no process in the process group indicated by *pgrp_id* that has the same controlling terminal as the calling process.

8. C Language Library

8.1 Referenced C Language Routines

When the *ANSI/X3.159-198x Programming Language C Standard* is adopted, it will be the basis for a C language binding to IEEE Std 1003.1. In the interim, the following routines are left unstandardized, but are defined by common usage and traditional implementations. Although the lack of an adopted C language standard negatively affects the ability of applications developers to write portable applications, they can use draft versions of the *ANSI/X3.159-198x Programming Language C Standard* and common usage as guidance to maximize the future portability of their applications.

- 4.2 Diagnostics
  Functions: assert.
- 4.3 Character Handling
  Functions: isalnum, isalpha, iscntrl, isdigit, isgraph, islcwer, isprint, ispunct, isspace, isupper, isxdigit, tolower, toupper.
- 4.5 Mathematics
  Functions: acos, asin, atan, atan2, cos, sin, tan, cosh, sinh, tanh, exp, frexp, ldexp, log, log10, modf, pow, sqrt, ceil, fabs, floor, fmod.
- 4.6 Non-Local Jumps
  Functions: setjmp, longjmp.
- 4.7 Signal Handling
  Functions: signal+.
- 4.9 Input/Output
  Functions: clearerr, fclose, feof, ferror, fflush, fgetc, fgets, fopen, fputc, fputs, fread, freopen, fseek, ftell, fwrite, getc, getchar, gets, perror, printf, fprintf, sprintf, putc, putchar, puts, remove, rename+, rewind, scanf, fscanf, sscanf, setbuf, tmpfile, tmpnam, ungetc.
- 4.10 General Utilities
  Functions: abs, atof, atoi, atol, rand, srand, calloc, free, malloc, realloc, abort, exit, getenv+, bsearch, qsort, setlocale+.

- 4.11 String Handling
  Functions: strcpy, strncpy, strcat, strncat, strcmp, strchr, strcspn, strpbrk, strrchr, strspn, strstr, strtok, strlen.

- 4.12 Date and Time
  Functions: time+, asctime+, ctime+, gmtime+, localtime+, strftime+.

Functions indicated above with a + are included in both documents. Descriptions of these routines have been retained in this standard because they represent further specifications or amplifications of the versions defined by the *ANSI/X3.159-198x Programming Language C Standard.*

Systems conforming to the IEEE Std 1003.1 shall make no distinction between the "text streams" and the "binary streams" described in the *ANSI/X3.159-198x Programming Language C Standard.*

For the *fseek*() function, if the specified position is beyond end-of-file, the consequences described in *lseek*() (see *lseek*() §6.5.3) shall occur.

8.1.1 Extensions to *asctime*() Function

If the environment variable named TZ is present, (see **Environment Variables** §2.7) the functions *asctime*(), *strftime*(), *localtime*(), *ctime*(), and *gmtime*() use its contents to override the default time zone. The value of TZ has the form (spaces inserted for clarity):

std offset dst offset ,*rule*

or in an expanded format:

stdoffset[*dst*[*offset*][,*start*[/*time*],*end*[/*time*]]]
/*characters*

Where:

If the first character of the environment variable TZ is a slash (/), it is assumed the characters following the slash are handled in an implementation defined manner.

std and *dst*
: Three or more bytes that are the designation for the standard (*std*) or summer (*dst*) time zone. Only *std* is required; if *dst* is missing, then summer time does not apply in this locale. Upper- and lowercase letters are explicitly allowed. Any characters except digits, comma (,), minus (–), plus (+), and ASCII NUL are allowed.

offset
: Indicates how far west (or, if preceded by "–", east) of Greenwich that time zone lies. The *offset* has the form:

hh [:*mm* [:*ss*]]

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) shall be required and may be a single digit. The *offset* following *std* shall be required. If no *offset* follows *dst*, summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour shall be between **0** and **12**, and the minutes (and seconds) — if present — between **0** and **59**. Out of range values may cause unpredictable behavior. If preceded by a "–", the time zone shall be east of Greenwich, otherwise it shall be west (which may be indicated by an optional preceding "+").

rule

Indicates when to change to and back from summer time. The *rule* has the form:

date / *time*,*date* / *time*

where the first *date* describes when the change from standard to summer time occurs and the second *date* describes when the change back happens. Each *time* field describes when, in current local time, the change to the other time is made.

The format of *date* shall be one of the following:

J*n*

The Julian day *n* ($1 \le n \le 365$). Leap days shall not be counted. That is, in all years — including leap years — February 28 is day 59 and March 1 is day 60. It is impossible to explicitly refer to the occasional February 29.

n

The zero-based Julian day ($0 \le n \le 365$). Leap days shall be counted, and it is possible to refer to February 29.

M*m***.***n***.***d*

The d^{th} day ($0 \le d \le 6$) of week *n* of month *m* of the year ($1 \le n \le 5$, $1 \le m \le 12$, where week 5 means "the last *d* day in month *m*" which may occur in either the fourth or the fifth week).

The *time* has the same format as *offset* except that no leading sign ("–" or "+") shall be allowed. The default, if *time* is not given, shall be

02:00:00.

If no *rule* is specified and summer time applies, United States federal law shall be assumed.

If the first character of the *rule* is a slash (/), the bytes following the slash shall be handled in an implementation defined manner.

The effects of setting **TZ** are, thus, to change the values of the external variable **timezone** and **daylight**. In addition, the time zone names contained in the external variable

```
char *tzname[2] = {"std","dst"};
```

are set from the environment variable **TZ**.

It is explicitly allowed for programs to change **TZ** and have the changed **TZ** apply to themselves.

8.1.2 Extensions to *setlocale*() Function

Function: *setlocale*()

8.1.2.1 Synopsis

```
char *setlocale (category, locale)
int category;
char *locale;
```

8.1.2.2 Description

The *ANSI/X3.159-198x Programming Language C Standard* allows the specification of an implementation defined native environment for the *setlocale*() function, which will set a specific category to an implementation defined default. For IEEE Std 1003.1 systems, this corresponds to the value of the environment variables.

Setting a specific category to an implementation defined default is invoked by setting the *locale* argument to point to a null string, and by setting the *category* argument to one of the integer values:

LC_CTYPE
LC_COLLATE
LC_TIME
LC_NUMERIC

In all cases, *setlocale*() will first check the value of the corresponding environment variable (e.g., LC_CTYPE for the LC_CTYPE category) and if valid (i.e., points to the name of a valid locale), *setlocale*() will set the specified category of the international environment to that value and return the string corresponding to the locale set (i.e., the value of the environment variable, not ""). If the value is invalid, *setlocale*() will return a null pointer and the international environment is not changed.

If the environment variable corresponding to the specified category is not set or is set to the empty string, the behavior of *setlocale*() is implementation defined, unless the LANG environment variable is set and valid in which case *setlocale*() will set the category to the corresponding value of LANG. In some implementations, this may default to a system-wide value, others may default to the "C" locale. Setting all categories to the implementation defined default is similar to the previous usage, but it interrogates all the environment variables to determine the specific value to set. To set all categories in the international environment, *setlocale*() is invoked in the following manner:

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

To satisfy this request, *setlocale*() first checks all the environment variables. If any environment variable is invalid, *setlocale*() returns a null pointer and the international environment is not changed.

If they are valid, *setlocale*() sets the international environment to reflect the values of the environment variables. The categories are set in the following order:

```
LC_ALL
LC_CTYPE
LC_COLLATE
LC_TIME
LC_NUMERIC
```
new categories

Using this scheme, the categories corresponding to the environment variables will override the value of the LANG environment variable for a particular category.

If one or all of the category-specific environment variables (i.e., LC_CTYPE, LC_COLLATE, LC_TIME, or LC_NUMERIC) are not set, the particular category is not overridden. If one or all of the category-specific environment variables are set to the empty string, the behavior is implementation defined.

If the LANG environment variable is not set or is set to the empty string, the behavior of *setlocale*() is implementation defined.

8.2 FILE-Type C Language Functions

This section describes functions which make reference to the FILE type, as described in the *ANSI/X3.159-198x Programming Language C Standard.*

8.2.1 Map a Stream Pointer to a File Descriptor

Function: *fileno*()

8.2.1.1 Synopsis

```
#include <stdio.h>

int fileno (stream)
FILE *stream;
```

8.2.1.2 Description

The *fileno*() function returns the integer file descriptor associated with the *stream* (see *open*() §5.3.1).

There is a fixed relationship between the C language *stdin*, *stdout*, and *stderr* and the initial corresponding file descriptor values. The following symbolic values in <unistd.h> §2.10 define this relationship:

STDIN_FILENO	Standard input value, *stdin*.
STDOUT_FILENO	Standard output value, *stdout*.
STDERR_FILENO	Standard error value, *stderr*.

8.2.1.3 References

open() §5.3.1.

8.2.2 Open a Stream on a File Descriptor

Function: *fdopen*()

8.2.2.1 Synopsis

```
#include <stdio.h>

FILE *fdopen (fildes, type)
int fildes;
char *type;
```

8.2.2.2 Description

The *fdopen*() routine associates a stream with a file descriptor.

The *type* argument is a character string having one of the following values:

`"r"`	open for reading
`"w"`	open for writing
`"a"`	open for writing at end of file
`"r+"`	open for update (reading and writing)
`"w+"`	open for update (reading and writing)
`"a+"`	open for update (reading and writing) at end of file

The types **r+**, **w+**, and **a+** are equivalent, except that **a+** implicitly seeks to the end of the file.

Additional values for the *type* argument may be defined by an implementation.

The *type* of the stream must be allowed by the mode of the open file.

8.2.2.3 Returns

If successful, the *fdopen*() function returns a pointer to a stream. Otherwise, a NULL pointer is returned.

8.2.2.4 References

open() §5.3.1, *fopen*() (*ANSI/X3.159-198x Programming Language C Standard*). 8

8.3 Other C Language Functions

8.3.1 Non-Local Jumps

Functions: *sigsetjmp*(), *siglongjmp*()

8.3.1.1 Synopsis

```
#include <setjmp.h>

int sigsetjmp (env, savemask)
sigjmp_buf env;
int savemask;

void siglongjmp (env, val)
sigjmp_buf env;
int val;
```

8.3.1.2 Description

The *sigsetjmp*() macro shall comply with the definition of the *setjmp*() macro in the *ANSI/X3.159-198x Programming Language C Standard.* If the value of the *savemask* argument is not zero, the *sigsetjmp*() function shall also save the process's current signal mask (see <signal.h> §3.3.1) as part of the calling environment.

The *siglongjmp*() function shall comply with the definition of the *longjmp*() function in the *ANSI/X3.159-198x Programming Language C Standard.* If and only if the *env* argument was initialized by a call to the *sigsetjmp*() function with a non-zero *savemask* argument, the *siglongjmp*() function shall restore the saved signal mask.

8.3.1.3 References

sigaction() §3.3.4, <signal.h> §3.3.1, *sigprocmask*() §3.3.5, *sigsuspend*() §3.3.7.

8.3.2 Specify Signal Handling

Function: *signal*()

8.3.2.1 Synopsis

```
#include <signal.h>

void (*signal (sig,func) ) ( )
int sig;
void (*func) ( );
```

8.3.2.2 Description

The *ANSI/X3.159-198x Programming Language C Standard* defines the *signal*() function as a means of specifying the action to be taken upon receipt of a signal.

In general, the use of the *signal*() function shall not conflict with the behavior of signals as characterized in this standard. However, there may be implementation defined side effects associated with the use of the *signal*() function. For instance, if the *signal*() function is invoked to establish a signal-catching function or to set the action to SIG_DFL while the signal is pending, the pending signal may be discarded (unless the signal is SIGKILL or SIGSTOP).

The *sigaction*() §3.3.4 function provides an alternative interface that assures the delivery of signals and the integrity of signal-catching functions.

The *sigaction*() function shall properly return, in the structure pointed to by *oact*, the previous signal action, even if that action had been established by the *signal*() function. In such a case, the values of the fields of the structure pointed to by *oact* are undefined, and in particular *oact->sv_handler* is not necessarily the same value passed to the *signal*() function. However, if a pointer to the structure is passed to a subsequent call to the *sigaction*() function via the *act* parameter, handling of the signal shall be reinstated as if the original call to the *signal*() function were repeated.

It is implementation defined whether the return value of the *signal*() function will accurately reflect the previous signal action if that action had been established by the *sigaction*() function. It is also implementation defined whether a signal mask established by the *sigaction*() function is preserved when the signal action for that signal is altered by the *signal*() function. Because of this unpredictability, the *sigaction*() and *signal*() functions should not be used in the same process to control the same signal.

9. System Databases

9.1 System Databases

The routines described in this section allow an application to access the two system databases that are described below.

The *group* database contains the following information for each group:

- group name
- numerical group ID
- list of the names or numbers of all users allowed in the group

The *passwd* database contains the following information for each user:

- login name
- numerical user ID
- numerical group ID
- initial working directory
- initial user program

If the initial program field is null, the system default is used.

If the initial working directory field is null, the interpretation of that field is implementation defined.

These databases may contain other fields that are implementation defined.

9.2 Database Access

9.2.1 Group Database Access

Functions: *getgrent*(), *getgrgid*(), *getgrnam*(), *setgrent*(), *endgrent*()

9.2.1.1 Synopsis

```
#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (gid)
uid_t gid;

struct group *getgrnam (name)
char *name;

void setgrent ( )

void endgrent ( )
```

9.2.1.2 Description

The *getgrent*(), *getgrgid*() and *getgrnam*() routines each return pointers to an object of type *struct group* containing an entry from the group database. The members of this structure, which is defined in <grp.h>, include:

Member Type	Member Name	Description
char *	gr_name	The name of the group
uid_t	gr_gid	The numerical group ID
char **	gr_mem	A null-terminated vector of pointers to the individual member names

The *getgrent*() function reads the next entry of the database, so successive calls shall search the entire database. The *getgrgid*() and *getgrnam*() functions search from the beginning of the database until a matching *gid* or *name* is found, or the end of the database is encountered.

A call to *setgrent*() has the effect of rewinding the group database to allow repeated searches. A call to the *endgrent*() function should be used to close the group database when processing is complete.

9.2.1.3 Returns

A NULL pointer is returned on error or when the end of the database is encountered.

The return values may point to static data that is overwritten by each call. 9

9.2.1.4 References

getlogin() §4.2.4, *getpwent*() §9.2.2.

9.2.2 User Database Access c

Functions: *getpwent*(), *getpwuid*(), *getpwnam*(), *setpwent*(), *endpwent*()

9.2.2.1 Synopsis

```
#include <pwd.h>

struct passwd *getpwent ()

struct passwd *getpwuid (uid)
uid_t uid;                                  8

struct passwd *getpwnam (name)
char *name;

void setpwent ()

void endpwent ()
```

9.2.2.2 Description

The *getpwent*(), *getpwuid*() and *getpwnam*() functions each return a pointer to an object of type *struct passwd* containing an entry from the user database. The members of this structure, which is defined in **<pwd.h>**, include: c

Member Type	Member Name	Description	
char *	pw_name	User's login name	B
uid_t	pw_uid	User ID number	8
uid_t	pw_gid	Group ID number	8
char *	pw_dir	Home Directory	
char *	pw_shell	Default shell	

The *struct passwd* structure used by these routines may include additional members. The additional member names shall be declared in **<pwd.h>** and shall begin with the prefix "pw_". 9 9 9

The *getpwent*() function reads the next entry in the database, so successive calls can be used to search the entire database. The *getpwuid*() and *getpwnam*() functions search from the beginning of the database until a matching *uid* or *name* is found, or the end of the database is encountered.

A call to *setpwent*() has the effect of rewinding the user database to allow repeated searches. A call to *endpwent*() closes the password database when processing is complete.

The implementation of the *cuserid*() §4.2.4 function may use the *getpwnam*() function; thus the results of a user's call to either routine may be overwritten by a subsequent call to the other routine.

9.2.2.3 Returns

A NULL pointer is returned on error or the end of the database is encountered.

The return values may point to static data that is overwritten on each call.

9.2.2.4 References

cuserid() §4.2.4, *getlogin*() §4.2.4, *getgrent*() §9.2.1.

10. Data Interchange Format

10.1 Archive/Interchange File Format

A conforming system shall provide a mechanism to copy files from a medium to the local file system and copy files from the local file system to a medium using the interchange format described here. This standard does not define this mechanism.*

When this mechanism is used to copy files from the medium by a nonprivileged process, the protection information (ownership and access permissions) shall be set in the same fashion that *creat*() §5.3.2 would when given the mode argument matching the file permissions supplied by the *mode* field of this format.

The *format-creating utility* is used to translate from the file system to the formats defined in this section, in an implementation defined way, and the *format-reading utility* is used to translate from the formats defined in this section to a file system.

10.1.1 `cpio` Archive Format

The byte-oriented `cpio` archive format is a series of entries, each comprised of a header that describes the file, the name of the file, and then the contents of the file.

An archive may be recorded as a series of fixed size blocks of bytes. This blocking shall be used only to make physical I/O more efficient. The last group of blocks is always at the full size.

For the byte-oriented `cpio` archive format, the individual entry information must be in the order indicated and is described by:

---

* The P1003.2 Working Group is working on this mechanism. See **Shell and Utilities** §A.2.2.

Byte-Oriented cpio Archive Entry

Header

Field Name	Length	Interpreted as
c_magic	6 bytes	octal number
c_dev	6 bytes	octal number
c_ino	6 bytes	octal number
c_mode	6 bytes	octal number
c_uid	6 bytes	octal number
c_gid	6 bytes	octal number
c_nlink	6 bytes	octal number
c_rdev	6 bytes	octal number
c_mtime	11 bytes	octal number
c_namesize	6 bytes	octal number
c_filesize	11 bytes	octal number

File Name

Field Name	Length	Interpreted as
c_name	c_namesize	pathname string

File Data

Field Name	Length	Interpreted as
c_filedata	c_filesize	data

10.1.1.1 Header

For each file in the archive, a header as defined above shall be written. The information in the header fields shall be written as streams of bytes interpreted as octal numbers and shall be right-justified and zero filled. The fields shall be interpreted as follows:

- *c_magic* shall identify the archive as being a transportable archive by containing the magic bytes as defined by MAGIC ("070707").
- *c_dev* and *c_ino* shall contain values which uniquely identify the file within the archive (i.e., no files shall contain the same pair of *c_dev* and *c_ino* values unless they are links to the same file). The values shall be determined in an implementation defined manner.
- *c_mode* shall contain the file type and access permissions as defined in the tables below.
- *c_uid* shall contain the user id of the owner.
- *c_gid* shall contain the group id of the group.
- *c_nlink* shall contain the number of links referencing the file at the time the archive was created.

- *c_rdev* shall contain implementation defined information for character or block special files.
- *c_mtime* shall contain the latest time of modification of the file.
- *c_namesize* shall contain the length of the path name, including the terminating null byte.
- *c_filesize* shall contain the length of the file. This is the length of the data section following the header structure.

10.1.1.2 File Name

c_name shall contain the path name of the file. The length of the name is determined by *c_namesize*; the maximum length of this string is 256 bytes.

10.1.1.3 File Data

Following *c_name*, there shall be *c_filesize* bytes of data. Interpretation of such data shall occur in a manner dependent on the file. If *c_filesize* is zero, no data shall be contained in *c_filedata*.

10.1.1.4 Special Entries

Special files, directories, and the trailer are recorded with *c_filesize* equal to zero. The header for the next file entry in the archive shall be written directly after the last byte of the file entry preceding it. A header denoting the file name "`TRAILER!!!`" shall indicate the end of the archive; the contents of bytes in the last block of the archive following such a header are undefined.

10.1.1.5 `cpio` Values

Values needed by the `cpio` archive format are described as follows:

Values for c_mode field

File permissions

Name	Value	Indicates
C_IRUSR	000400	read by owner
C_IWUSR	000200	write by owner
C_IXUSR	000100	execute by owner
C_IRGRP	000040	read by group
C_IWGRP	000020	write by group
C_IXGRP	000010	execute by group
C_IROTH	000004	read by others
C_IWOTH	000002	write by others
C_IXOTH	000001	execute by others
C_ISUID	004000	set uid
C_ISGID	002000	set gid
C_ISVTX	001000	reserved

Values for c_mode field

File type

Name	Value	Indicates
C_ISDIR	040000	directory
C_ISFIFO	010000	FIFO
C_ISREG	100000	regular file
C_ISBLK	060000	block special
C_ISCHR	020000	character special
	110000	reserved
	120000	reserved
	140000	reserved

C_ISDIR, C_ISFIFO, and C_ISREG shall be supported on a IEEE Std 1003.1 conforming system; additional values defined above are reserved for compatibility with existing systems. Additional file types may be supported; however, such files should not be written on archives intended for transport to portable systems.

10.1.1.6 References

<grp.h> §9.2.1, <pwd.h> §9.2.2, <sys/stat.h> §5.6.1, *chmod*() §5.6.4, *link*() §5.3.4, *mkdir*() §5.4.1, *read*() §6.4.1, *stat*() §5.6.2.

10.1.2 Multiple Volumes

It shall be possible for data represented by the Archive/Interface File Format to reside in more than one file.

The format is considered a stream of bytes. Any two bytes may be separated by the end of a file.

The end-of-file is used as an indicator that a new file is to be read, and the format-reading utility will, in an implementation defined manner, determine the next file.

Appendices

(These appendices are not a part of IEEE Std 1003.1, IEEE Standard Portable Operating System Interface for Computer Environments.)

A. Related Standards

This appendix describes other standards efforts, related to IEEE Std 1003.1, that are available or under development.

A.1 Related Standards — Open System Architecture

This IEEE Std 1003.1 is intended to complement others that together would provide a comprehensive Open System Architecture. The standards in these areas fall into three areas: ones directly related to the IEEE Std 1003.1, ones already available and of use to those interested in Open Systems Architectures, and finally, those in development.

IEEE and ANSI/IEEE standards can be ordered from:

IEEE Service Center	IEEE Computer Society
445 Hoes Lane	Box 80452, Worldway Postal Center
Piscataway, NJ 08854	Los Angeles, CA 90080
(201) 981-0060	(800) 272-6657
	(714) 821-8380 in California

The document X3/SD-4 provides a list of all active X3 and related ISO projects, including approved standards. X3/SD-4 is available from:

CBEMA
X3 Secretariat
311 First Street, NW Suite 500
Washington, DC 20001-2178
(202) 737-8888

ANSI and ISO standards can be ordered from:

ANSI
1430 Broadway
New York, NY 10018
(212) 642-4900

A.2 Standards Closely Related to the 1003.1 Document

A.2.1 C Language Standard

This document refers to the C Language Standard effort presently under development by Technical Committee X3J11 of the Accredited Standards Committee X3 — Information Processing Systems. The X3J11 and 1003.1 groups have been cooperating to insure that the standards are complementary and not overlapping. At the time of publication, the most recent X3J11 material was the version for public comment of the *ANSI/X3.159-198x Programming Language C Standard*, available from:

Global Engineering Documents, Inc.
2625 Hickory Street
Santa Ana, CA 92707
(800) 854-7179
(714) 540-9870

Once the X3J11 document is approved, it will be available from the ANSI address given above.

A.2.2 Shell and Utilities

This area is currently in development by IEEE Computer Society Working Group P1003.2. The proposed 1003.2 standard defines a source code level interface to shell services and common utility programs for application programs conforming to IEEE Std 1003.1.* The proposed standard is being designed to be used by both application programmers and system implementors.

The following goals have been established for the Working Group:

Specify a standard interface that may be accessed in common by both applications programs and user terminal-controlling programs to provide services of a more complex nature than the primitives provided by IEEE Std 1003.1. This interface shall be implementable on conforming IEEE Std 1003.1 systems. It shall include the following components:

1. Application program primitives to specify instructions to an implementation defined "shell" facility.

2. A standard command language for a shell that includes program execution, I/O redirection and pipelining, argument handling, variable substitution and

* An IEEE Std 1003.1 conforming *implementation* is not necessarily required to support these application programs. Implementations could be produced that are conformant only to those 1003.1 features required by the proposed 1003.2 standard, and that cannot claim full conformance to all of IEEE Std 1003.1.

expansion, and a series of control constructs similar to other high-level structured programming languages.

3. A recommended command syntax for command naming and argument specification.
4. Primitives to assist applications programs and the shell language in parsing and interpreting command arguments.
5. Recommended environment variables for use by shell scripts and application programs.
6. A minimum directory hierarchy required for the shell and applications.
7. A group of utilities that may be called from application programs for complex data manipulation and other tasks common to many applications. c c
8. An optional group of utilities to be used for the software development of applications. c c
9. Utilities and standards for the installation of applications. c

The following areas are outside the scope of this standard:

1. Operating system administrative commands (privileged processes, system processes, daemons, etc.).
2. Commands required for the installation, configuration, or maintenance of operating systems or file systems.*
3. Networking commands.
4. Terminal control or user-interface programs (visual shells, window managers, command history mechanisms, etc.).
5. Graphics programs or interfaces.
6. Text formatting programs or languages.
7. Database programs or interfaces (e.g. SQL, etc.).

At the time of this printing, no published document existed. Working drafts were being circulated, with a target schedule of early 1989 for balloting. c

* This is contrasted against paragraph i, above, by its orientation to installing the operating system itself, versus application programs. The exclusion of operating system installation facilities should not be interpreted to mean that the non-privileged application installation procedures *cannot* be used for installing operating system components.

If you are interested in participating in this effort contact the IEEE Standards Office; the address is listed in the Foreword.

A.2.3 Verification Testing

This area is currently in development by IEEE Computer Society Working Group P1003.3.

If you are interested in obtaining 1003.3-related documents, or in participating in this effort, contact the IEEE Standards Office.

A.2.4 Real Time Extensions

A project has been approved for IEEE Computer Society Working Group P1003.4 to develop and ballot extensions to IEEE Std 1003.1 to address service interfaces needed for portable real time applications. This working group is an outgrowth of the /usr/group Technical Committee Real Time Subcommittee. At the time of publication, no draft document existed.

Contact the IEEE Standards Office to participate in this effort.

A.2.5 Language Standards

The following language standards are available from ANSI:

Ada	Mil Std 1815-A-1983
Basic	X3.113-1987
Cobol	X3.23-1985
Fortran	X3.9-1978
Mumps	MDC X11.1-1984
Pascal	X3.97-1983

A.2.6 Networking Standards

The ISO/OSI (Open System Interconnect) networking specifications are available from CBEMA or ANSI (and 802.*n* from the IEEE Standards Office):

OSI Model		ISO 7498 (ANSI)
Layer 1	CSMA/CD	IEEE 802.3 (IEEE)
	Token Bus	IEEE 802.4 (IEEE)
	Token Ring	IEEE 802.5 (IEEE)
Layer 2	Link Layer Control	IEEE 802.2 (IEEE)
		CCITT DR X.212 (CBEMA)
Layer 3	Network Layer	ISO 8348, 8473, 7777 (CBEMA)
Layer 4	Transport Layer	ISO 8072, 8073 (CBEMA)

Layer 5	Session Layer	ISO 8326, 8327 (CBEMA)	c
Layer 6	Presentation Layer	ISO DP 8822, DP 8823 (CBEMA)	c
Layer 7	Applications Layer		c
	CASE (Common Services)	ISO DP 8649, DP 8650 (CBEMA)	c
	FTAM (File Transfer)	ISO DP 8571 (CBEMA)	c
	Mail/Message	CCITT X.400 series (CBEMA)	c
	Job Transfer	ISO DP 8831, DP 8832 (CBEMA)	c
Wide Area Net	Layers 1-3	CCITT X.25 (CBEMA)	c

A.2.7 Graphics Standards c

The following graphics-related standards are available from CBEMA or ANSI: c

GKS	X3.124-1985 *Graphical Kernel System*; C language bindings are in progress (0533-D). (ANSI)	c
PHIGS	X3.144-198*x* *Programmers' Hierarchical Interactive Graphics System*; C language bindings are in progress (0534-D). (CBEMA)	c
CGM	X3.122-1986 *Computer Graphics Metafile*, formerly known as VDM, Virtual Device Metafile. (CBEMA)	c
X3H3.6	This working group is addressing windowing standards and display management for graphical devices. (CBEMA)	c

A.2.8 Data Base Standards

The following data base standards are available from ANSI: c

NDL	X3.133-1986 *Database Language NDL*. (Network Databases.)	c
SQL	X3.135-1986 *Database Language SQL*. (Relational Databases.)	c

A.3 Industry Open Systems Publications

The following publications describe recommendations formed by industry groups (as opposed to a single company) about related standards efforts.

The *X/OPEN Portability Guide* is available from:

Elsevier Science Publishers Co. Inc,
P.O. Box 211
Grand Central Station,
New York, NY 10163

A.4 US Government Standards

A.4.1 Federal Information Processing Standards (FIPS)

The following standards are designated by the US Government as Federal Information Processing Standards. These frequently refer back to standards listed above. Information on these can be obtained from:

National Technical Information Service
US Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
(703) 487-4650

An index for FIPS standards is *NBS Publications List 58*, available as document number 301-975-2816.

A.4.2 Trusted Systems

A standard for secure, or trusted, systems, the *Department of Defense Trusted Computer System Evaluation Criteria*, Department of Defense Standard DoD 5200.28-STD, December 1985, is available from:

Office of Standards and Products
National Computer Security Center
Fort Meade, MD 20755-6000
Attn: Chief, Computer Security Standards

B. Rationale and Notes

This appendix summarizes the deliberations of the IEEE P1003.1 Working Group, the committee charged by IEEE with devising an interface standard for a portable operating system interface for computer environments, IEEE Std 1003.1.

This appendix is derived in part from copyrighted draft documents developed under the sponsorship of /usr/group*, as part of an ongoing program of that association to support the IEEE 1003 standards program efforts.

The appendix is being published along with the standard to assist in the process of review. It contains historical information concerning the contents of the standard and why features were included or discarded by the Working Group. It also contains notes of interest to application programmers on recommended programming practices, emphasizing the consequences of some aspects of the standard that may not be immediately apparent.

B.1 Introduction

The IEEE Std 1003.1 is based on the UNIX operating system developed by AT&T Bell Laboratories, and derives from efforts of the Standards Committee of /usr/group, an association of individuals, corporations, and institutions with an interest in the UNIX system that has long worked toward the development of independent industry-driven standards. The IEEE P1003 Working Group represents a cross-section of the UNIX system community: it consists of over 250 members representing hardware manufacturers, vendors of operating systems and other software development tools, software designers, consultants, academics, authors, applications programmers, and others. In the course of its deliberations, it has reviewed related American and international standards, both published and in progress. This revision includes responses and rationale material related to the comments received in the trial use period.

Although originally coined by the IEEE to refer to IEEE Std 1003.1, the term POSIX more correctly refers to a *family* of related standards or working groups, P1003.*n*. These other activities are described in Appendix A. There are some cases where this rationale uses the term POSIX as a synonym for IEEE Std 1003.1. This incorrect usage is maintained for purposes of readability only. The body of the standard does not use the term POSIX in this way.

As explained in the Foreword, the term POSIX is expected to be pronounced *pahz-icks*, as in *positive*, not *poh-six*, or other variations. The P1003 Working Group has published the pronunciation of its term in an attempt to promulgate a standardized way of referring to a standard operating system interface.

The intended audience for this standard is all persons concerned with an industry-wide standard operating system based on the UNIX system. This includes at least four groups of people:

1. persons buying hardware and software systems;
2. persons managing companies that are deciding on future corporate computing directions;
3. persons implementing operating systems, and especially;
4. persons developing applications where portability is an objective.

B.1.1 Scope

This Rationale focuses primarily on additions, clarifications, and changes made to the UNIX system as described in the **Base Documents** §B.1.3 from which the standard was derived. It is *not* a rationale for the UNIX system as a whole, since the Working Group was charged with codifying existing practice, not designing a new operating system. No attempt is made in this Rationale to defend the pre-existing structure of UNIX systems. It is primarily deviations from existing practice, as codified in the Base Documents, that are explained or justified here.

The Rationale discusses some UNIX system features that were *not* adopted into the standard. Many of these are features that are popular in some UNIX system implementations, so that a user of those implementations might question why they do not appear in the standard.

There are choices allowed by the standard for some details of the interface specification; some of these are specifiable option subsets of the standard. See **Portability Specifications** §B.2.10. See also **Specific Derivations** §B.1.3.3.

The standard is not a tutorial on the use of the specified interface, nor is this Rationale. However, the Rationale includes some references to well-regarded historical books on the UNIX System in **Historical Implementations** §B.11.2.

B.1.2 Purpose

Several principles guided the Working Group's decisions.

B.1.2.1 Application Oriented

The basic goal of the Working Group was to promote portability of application programs across UNIX system environments by developing a clear, consistent, and unambiguous standard for the interface specification of a portable operating system based on the UNIX system documentation. This standard codifies the common, existing definition of the UNIX system. There was no attempt to define a new system interface.

B.1.2.2 Interface, Not Implementation

The standard defines an interface, not an implementation. No distinction is made between library functions and system calls: both are referred to as functions. No details of the implementation of any function are given (although historical practice is sometimes indicated in the Rationale). Symbolic names are given for constants (such as signals and error numbers) rather than numbers.

B.1.2.3 Source, Not Object, Portability

The standard has been written so that a program written and translated for execution on one conforming implementation may also be translated for execution on another conforming implementation. The standard does not guarantee that executable (object) code will execute under a different conforming implementation than that for which it was translated, even if the underlying hardware is identical. The Working Group has, however, attempted to put few impediments in the way of binary compatibility, and some remarks are found in this Rationale. See **Requirements** §B.2.2.1.1 and **Configurable System Variables** §B.4.8.

B.1.2.4 The C Language and X3J11

The standard is written in terms of the standard C language as specified in the *ANSI/X3.159-198x Programming Language C Standard* that the X3J11 Working Group produced. See **Conformance** §2.2. Guidelines used in negotiations between the two Working Groups are discussed below in **C Language, X3J11, and P1003.1** §B.1.4.

B.1.2.5 No Super-User, No System Administration

There was no intention to specify all aspects of an operating system. System administration facilities and functions are excluded from the standard, and functions usable only by the super-user have not been included. This Rationale notes several such instances. Still, an implementation of the standard interface may also implement features not in the standard: see **Requirements** §2.2.1.1. The standard is also not concerned with hardware constraints or system maintenance.

B.1.2.6 Minimal Interface, Minimally Defined

In keeping with the historical design principles of the UNIX system, the standard is as minimal as possible. For example, it usually specifies only one set of functions to implement a capability. Exceptions were made in some cases where long tradition and many existing applications included certain functions, such as *creat*() §5.3.2. In such cases, as throughout the standard, redundant definitions were avoided: *creat*() §5.3.2 is defined as a special case of *open*() §5.3.1. Redundant functions or implementations with less tradition were excluded. For example, *seekdir*() §B.5.1.2 and *telldir*() §B.5.1.2 were not included in **Directory Operations** §5.1.2.

B.1.2.7 Broadly Implementable

The Working Group has endeavored to make all specified functions implementable across a wide range of existing and potential systems, including:

- All of the current major systems that are ultimately derived from AT&T code (Version 7 or later).
- Compatible systems that are not derived from AT&T code.
- Emulations hosted on entirely different operating systems.
- Networked systems.

- Distributed systems.
- Systems running on a broad range of hardware.

No direct references to this goal appear in the standard, but some results of it are mentioned in this Rationale.

B.1.2.8 Minimal Changes to Historical Implementations

There is no known **historical implementation** §B.2.3 that will not have to change in some area to conform to the standard, and in a few areas the standard does not exactly match any existing system interface (for example, see **O_NONBLOCK** §B.6). Nonetheless, there is a set of functions, types, definitions, and concepts that form an interface that is common to most historical implementations. The standard specifies that common interface and extends it in areas where there has historically been no consensus, preferably

1. by standardizing an interface like one in an historical implementation, e.g., **Directories** §5.1, or
2. by specifying an interface that is readily implementable in terms of, and backwards compatible with, existing implementations, such as **TAR** §10.1, or
3. by specifying an interface that, when added to a historical implementation, will not conflict with it, like **O_NONBLOCK** §B.6.

Required changes to historical implementations have been kept as few as possible, but they do exist, and this Rationale points out some of them.

The standard is specifically not a codification of a particular vendor's product. It is like the UNIX system, but it is not identical to it. The word UNIX is not used in the standard proper both for that reason, and because it is a trademark of a particular vendor.

B.1.2.9 Minimal Changes to Existing Application Code

The Working Group wished to make less work for application developers, not more. However, because every known historical implementation will have to change at least slightly to conform, some applications will have to change. This Rationale points out the major places where the standard implies such changes.

B.1.2.10 IEEE Consensus Process

The IEEE consensus process was used in deliberations. There are several levels of participation:

- **Correspondents.**
  Those interested in following the development of the standard could subscribe to a mailing list to which copies of drafts, working documents, and related material were sent. Also, anyone (including individuals, companies, government agencies, or other organizations) could send comments (or RFCs, Proposals, or Notes) to the Working Group.

- **Working Group.**
  This was the group responsible for producing the standard document. It met four times a year and produced many drafts. It also produced the Trial Use and Full Use Standards, and was responsible for resolving balloting objections to them. The Working Group was composed of individuals, even though many of them worked for companies with interests in the field.

- **Balloting Group.**
  This group voted on the proposed standards in the manner detailed in the next subsection. The Balloting Group, like the Working Group, was composed of individuals. Most of the people on the Working Group also were in the Balloting Group, although the latter included many others, as well.

- **Institutional Representatives.**
  Exceptions to the individual composition of the Balloting Group were the Institutional Representatives, who represented related standards bodies or professional organizations (in this case, USENIX, /usr/group, and X/OPEN). These Institutional Representatives also served on the Working Group, but participated there as individuals.

Decisions of the Working Group were not made by vote, not even of a large majority. Decisions were made by consensus, which required that each individual believe that

- their point of view had been heard
- their point of view had been understood
- other individuals' points of view were adequately understood
- there was general consensus.

A common way of moving discussion along was to ask if anyone would ballot "no" on a particular issue.

B.1.2.11 IEEE Balloting Process

The IEEE balloting process is used to attain the ANSI requirement for a consensus acceptance of a document as a standard.

Balloting in IEEE is done by individuals who are members of IEEE or affiliated with the IEEE Computer Society. They are given thirty days in which to return the ballots, and 75% of those in the balloting group must return ballots.

Ballots from non-IEEE members are also included in the process, with comments and objections treated the same as those from members. However, non-IEEE members are not included in the percentages of returns required or the affirmative percentage required for approval. Possible ballot responses [excluding abstentions] are:

- yes without comments

- **yes with comments**
  The comments indicate areas that should be evaluated, but are not significant enough to warrant a negative ballot.

- **no with objections**
  A negative ballot must include specific objections and recommendations on how to resolve the objections. These objections indicate areas that must be fixed to resolve the negative ballot.

At least 75% of those balloting [not abstaining] must provide an affirmative response. Each objection, and many of the comments, are translated into proposed changes; and any outstanding objections, along with the rationale for not making the changes to accommodate these objections, are fed back to the balloting group.

Members of the balloting group are given ten days to change their ballots, with similar options as above; however, objections are limited to the proposed changes and/or failure to resolve key objections. It is possible for the number of negative responses to increase if a proposed change is objectionable, or if a significant objection has not been addressed.

In general, the balloting process moves fairly quickly towards a high degree of consensus. The final results are submitted to the IEEE Standards Board for approval, and include the balloting percentages as well as documentation of any unresolved negative objections.

The Trial Use period was from April 1986 to the November 1987, when the balloting of the revised document [Draft 12] began, and provided an additional level of industry consensus. The high visibility of the document, as well as its widespread distribution, provided additional feedback and information for the formulation of the current standard. See also Specific Derivations §B.1.3.3.

The Institutional Representatives were exceptions in several ways.

- They are not required to be IEEE members.
- They ballot for their Institutions, not as individuals.
- Ballots of Institutional Representatives are reported separately to the IEEE Standards Board.

As with other ballots, any unresolved negative objections are reported with the rationale for not incorporating the associated changes. However, the separate reporting of the Institutional ballots tends to make any objections more visible, particularly in that Institution's areas of expertise; consequently, any unresolved objection could be enough to cause the document to be sent back to the balloting process for further resolution. USENIX balloted affirmative for the Trial Use Standard; /usr/group balloted negative, and their unresolved issue was mandatory locking; X/OPEN did not ballot.

B.1.3 Base Documents

The Working Group consulted a number of documents as representing features appropriate for consideration for inclusion in the standard. Full bibliographic information may be found in **Bibliographic Notes** §B.11.

B.1.3.1 Related Standards and Documents

- *1984 /usr/group Standard*
- *ANSI/X3.159-198x Programming Language C Standard*
- *X/OPEN Portability Guide*

The most direct ancestor is the *1984 /usr/group Standard*, which is considered to be Draft 1 of the present standard. It, in turn, was largely derived from the programming interface of System III. The *1984 /usr/group Standard* is also the principal ancestor of the Library section of the C Standard.

The X3J11 and P1003.1 Working Groups have cooperated closely. Details of the relations of the two standards they produced are listed in this Rationale in **C Language, X3J11, and P1003.1** §B.1.4 because the C Standard is the standard most closely related to POSIX. POSIX is written in terms of the C Standard, although it is possible to have POSIX without Standard C: see **Conformance** §B.2.2.

The *X/OPEN Portability Guide* proved useful because X/OPEN had in many cases already addressed the same issues as P1003.1, though often in a slightly different context.

The Working Group is aware of the Japanese SIGMA project, which includes as a goal a common operating system interface specification, and there has been a representative of SIGMA at most recent P1003.1 Working Group meetings.

B.1.3.2 Historical Implementations

These include (with colloquial names in parentheses):

- *UNIX Time-Sharing System: UNIX Programmer's Manual, Seventh Edition (Version 7)*
- *UNIX System III Programmer's Manual*
- *AT&T System V Interface Definition (SVID), Issue 2, Volumes 1-3*
- *4.3 Berkeley Software Distribution, Virtual VAX-11 Version (4.3BSD) Manuals*

The UNIX system has changed more since the *1984 /usr/group Standard* was written than has the C language, and there are more variants of the former. Because of this, the present standard has been radically reorganized and reformatted since the first draft and has had many changes in content. Thus there is no single Base Document to provide context for all discussions in this Rationale, which instead discusses aspects of Version 7, System III, System V, and 4.3BSD that were included in this standard or that were considered in choosing what was included.

Occasional mentions are made of Version 8 and Version 9, which are successors of Version 7, the Bell Laboratories research system. The context is usually related to the *streams* inter-process communication mechanism, which is not in this standard but which has influenced discussions about inter-process communication mechanisms.

Although 4.2BSD was the current Berkeley Software Distribution when most of the work on the standard was done, this Rationale refers to 4.3BSD instead (in most places) because the differences between the two versions are almost entirely in performance, the few programming interface differences are mostly outside the scope of this standard, and the 4.3BSD manuals actually describe 4.2BSD better than the 4.2BSD manuals do.

The System V manuals are never referenced because the *SVID* is more definitive.

Much of the standard is closer to the *SVID* than to any other document, and there is an appendix that compares the two directly.

Parts of documentation of many other related systems were considered in deliberations on various aspects of the standard. As those were too numerous to list all of them, none of them will be mentioned by name.

B.1.3.3 Specific Derivations

Some areas of the standard are clearly derived from facilities of specific systems. Most of the major areas are listed here, together with references to the sections of the standard where they occur. For most of them, there is also more detail in the corresponding sections of the Rationale.

FIFOs
: The **FIFO special file** §2.3 facility exists in System III, the *1984 /usr/group Standard*, and System V, but not in Version 7, 4.2BSD, or 4.3BSD.

reliable signals
: **Signals** §3.3 includes reliable signals related to the 4.3BSD model. These were introduced between the Trial Use and Full Use Standards.

job control
: The **job control** §B.3.3 facility is derived from 4.3BSD and was introduced between the Trial Use and Full Use Standards.

saved set-user-ID (saved set-group-ID)
: This optional capability, mostly in *exec* §3.1.2 and **Set User and Group IDs** §4.2.2, is derived from System V, and was introduced in the Trial Use Standard.

supplementary groups
: A single group per process as in System V is the default, but **User**

Identification §4.2 (particularly *getgroups*() §4.2.3) allows multiple groups per process as in 4.3BSD as an option. This was introduced shortly before the Trial Use Standard.

uname()

The *uname*() §4.4.1 function is derived from the *1984 /usr/group Standard*, which took it from System III, and it is still in System V. It does not exist in Version 7 or 4.3BSD.

opendir(), *readdir*(), *rewinddir*(), *closedir*()

Directory Operations §5.1 is derived from 4.2BSD and was introduced in an early draft of the standard. It was later adopted in System V Release 3.

mkdir(), *rmdir*(), *rename*()

The three functions *mkdir*() §5.4.1, *rmdir*() §5.5.2, and *rename*() §5.5.3 are derived from 4.2BSD. Except for *rename*(), these functions now also appear in System V Release 3.

termios

Device- and Class-Specific Functions §7, while closer to System V than to 4.3BSD, does not correspond to any existing system because none was found adequate when considerations such as international character sets, fast interfaces, and networks were taken into account. The final interface specification was introduced shortly before the Full Use Standard.

archive format

The **Extended tar Format** §D.1 is derived from the *tar* programs used in Version 7 and 4.3BSD, and provided with System V. The precise format in the Full Use Standard has evolved incrementally from that in earlier drafts of POSIX.

B.1.3.4 Working Documents

The model for the present Rationale was the Rationale prepared by the X3J11 Working Group to accompany the *ANSI/X3.159-198x Programming Language C Standard*:

- *X3J11/86-152, October 1, 1986* "Rationale for Draft Proposed American National Standard for Information Systems—Programming Language C"

Its influence may be seen most clearly in **C Language, X3J11, and P1003.1** §B.1.4, but it also is present in more subtle ways throughout.

References to programs, functions, or facilities of systems described by the Base Documents (such as the System V `cpio` utility program) have been freely included in this Rationale where relevant, even though they would be inappropriate in the standard itself. References to programs, functions, or facilities not described by the base documents or to companies not directly associated with them have been excluded where possible. Exceptions have been made where facilities were derived from systems not described by the base documents, and where the word "may" is used to describe an option that permits behavior of such a system.

B.1.4 C Language, X3J11, and P1003.1

Some C language functions and definitions were handled by P1003.1, but most by X3J11. The most general guideline was that P1003.1 retained responsibility for operating-system specific functions, while X3J11 defined C library functions. See also **C Language Definitions** §B.2.8 and **C Language Library** §B.8..

There are several areas in which the two standards differ philosophically:

- **Function parameter type lists.**
  These appear in the C Standard and specify the types of the arguments and return values of functions in external references to them. POSIX does not include them, except in a few places to indicate variable number of arguments, e.g., **File Control** §B.6.5.2. Function parameter type lists were not used because the Working Group was aware that some vendors would wish to implement POSIX in terms of a binding to an historical variant of the C language instead of to the *ANSI/X3.159-198x Programming Language C Standard*, since compilers for the latter would initially not be widespread. Since the C Standard does not require the use of function parameter type lists, the function definitions used in POSIX are nonetheless specified in terms of Standard C. POSIX implementors whose C implementations support ANSI-style function prototypes should consider using them for declarations in POSIX. (Note that some code with improper declarations may have problems if this is done.) See also *signal*() §B.3.3.3.

- **Single vs. multiple processes.**
  The C Standard specifies a language that can be used on single-process operating systems and as a freestanding base for the implementation of operating systems or other stand-alone programs. But the POSIX interface is that of a multi-process timesharing system. Thus POSIX has to take multiple processes into account in places where the C Standard does not mention processes at all, such as *kill*() §3.3.2. See also **Requirements** §B.2.2.1.1.

- **Single vs. multiple operating system environments.**
  The C Standard specifies a language that may be useful on more than one operating system, and thus has means of tailoring itself to the particular current environment. POSIX is an operating system interface specification, and thus by definition is only concerned with one operating system environment, even though it has been carefully written to be **broadly implementable** §B.1.2.7 in terms of various underlying operating systems. See also **Requirements** §B.2.2.1.1.

- **Translation vs. execution environment.**
  POSIX is primarily concerned with the Standard C execution environment, leaving the translation environment to the C Standard. See also Requirements §B.2.2.1.1.

- **Hosted vs. freestanding implementations.**

  All POSIX implementations are hosted in the sense of the C Standard. See also the remarks on conformance in the **Foreword**.

- **Text vs. binary file modes.**
  X3J11 defines "text" and "binary" modes for a file. But the POSIX interface and historical implementations related to it make no such distinction, and all functions defined by P1003.1 treat files as if these modes are identical. (It is important not to say that POSIX files are either "text" or "binary.") X3J11 wrote their definitions so that this interpretation is possible. In particular, "text" mode files are not required to end with a line separator, which also means that they are not required to include a line separator at all.

And there is a basic difference in approach between the X3J11 Rationale and the P1003.1 Rationale. The X3J11 Rationale addresses almost all changes as differences from the Base Documents of the C Standard, usually either Kernighan and Ritchie or the *1984 /usr/group Standard*. The present Rationale cannot do that, since there are many more variants of (and Base Documents for) the operating system interface than for the C language. The most noticeable aspect of this difference is that X3J11 marks QUIET CHANGES from the Base Documents in its Rationale. The POSIX Rationale cannot include such markings, since a quiet change from one historical implementation may correspond exactly to another historical implementation, and may be very noticeable to an application written for yet another.

B.1.4.1 Solely by P1003.1.

These return parameters from the operating system environment: *cuserid*() §4.2.4, *ctermid*() §4.7.1, *ttyname*() §4.7.2, and *isatty*() §4.7.2.

The functions *fileno*() §8.2.1 and *fdopen*() §8.2.2, map between C Language stream pointers and POSIX file descriptors.

B.1.4.2 Solely by X3J11.

There are many functions that are useful with the operating system interface and are required for conformance with the present standard, but that are properly part of the C Language. These are listed in Referenced C Language Routines §8.1, which also notes which functions are defined by both P1003.1 and X3J11. Certain terms defined by X3J11 are incorporated by P1003.1 in C Language Definitions §2.8.

Some routines were considered too specialized by the P1003.1 Working Group to be included in the standard. These include *bsearch*() and *qsort*().

B.1.4.3 By Neither P1003.1 nor X3J11.

Some functions were considered of marginal utility and problematical when international character sets were considered: *_toupper*(), *_tolower*(), *toascii*(), and *isascii*().

Though *malloc*() §8.1 and *free*() §8.1 are in the C Standard and are required by **Referenced C Language Routines** §8.1 of the present standard, neither *brk*() §B.1.4.3 nor *sbrk*() §B.1.4.3 occur in either standard (although they were in the *1984 /usr/group Standard*), because this standard is designed to provide the basic set of functions required to write a Conforming Application; the underlying implementation of *malloc*() or *free*() is not an appropriate concern for the standard.

B.1.4.4 Base by P1003.1, Additions by X3J11.

Since the C Standard does not depend on POSIX in any way, there are no items in this category.

B.1.4.5 Base by X3J11, Additions by P1003.1.

X3J11 has to define *errno* if only because examining that variable is the only way to tell when some mathematics routines fail. But P1003.1 uses it more extensively, and adds some semantics to it in **Error Numbers** §2.5, which also defines some values for it.

Many numerical limits used by X3J11 were incorporated by P1003.1 in **Numerical Limits** §2.9, and some new ones are added, all to be found in the header **<limits.h>**.

The semantics of arguments to *main*() §3.1.2 are only defined in POSIX.

The POSIX definition of *signal*() §8.3.2 further specifies the C definition, and the entire mechanism of *signals* §3.3 is much more elaborate.

The function *time*() §4.5.1 is used by X3J11, but POSIX further specifies the time value.

The function *getenv*()4.6.1 is referenced in **Environment Description** §2.7 and *exec* §3.1.2 and is also defined by X3J11.

The function *rename*() §5.5.3 is extended to further specify its behavior when the new filename already exists or either argument refers to a directory.

B.1.4.6 Related Functions by Both.

The X3J11 definition of compliance and the P1003.1 definition of **Conformance** §2.2 are similar, although the latter notes certain potential hardware limitations.

P1003.1 defined a portable filename character set in **General Terms** §2.3, that is like the X3J11 identifier character set. However, P1003.1 did not allow upper- and lowercase characters to be considered equivalent. See **filename portability** §2.4.

The type *clock_t* §2.6 appears in both standards. See *Time* §B.4.5.

The *exit*() function is defined only by X3J11, because it refers to closing streams, and that subject, as well as *fclose*() itself, is defined almost entirely by X3J11. But P1003.1 defined *_exit*() §3.2.2, which also adds semantics to *exit*(). This also allows POSIX to ignore the X3J11 *atexit*() function.

P1003.1 defined *kill*() §3.3.2, while X3J11 defined *raise*(), which is similar except that it does not have a process ID argument, since the language defined by X3J11 does not incorporate the idea of multiple processes.

The new functions *sigsetjmp*() §8.3.1 and *siglongjmp*() §8.3.1 were added to provide similar functions to X3J11 *setjmp*() and *longjmp*() that additionally save and restore signal state. Requiring *setjmp*() and *longjmp*() to do this would have conflicted with the X3J11 definitions.

B.1.5 Organization

B.1.5.1 Organization of the Standard

See the **Foreword**.

It was decided very early that the traditional organization by manual section, as used in the *1984 /usr/group Standard*, would be confusing in an IEEE standard. That organization assumed some background that was not relevant to the purpose of the standard. It also made an implementation-oriented distinction between system calls and library routines, which were in separate sections.

Two sections, **Scope** §1 and **Definitions** §2, have been prepended because they are traditional in IEEE standards. A **Foreword** was prepended for the same reason, even though it is not part of the standard proper. The name POSIX, suggested by Richard Stallman, was adopted during the printing of the Trial Use Standard.

Although appendices were used in the Trial Use Standard to contain proposals for examination by the Balloting Group and the general public, the Full Use Standard has no proposal appendices, because the text of the standard proper must be complete. The Appendices of the Full Use Standard discuss either related standards or the Full Use Standard itself. *Editor's Note: Appendices D and E are an exception to the preceding two sentences. They will not appear in the Full Use Standard after it is approved, being included only to expedite the balloting process.* The Full Use Standard contains some new material that was not in the Trial Use Standard, mostly that which was added to meet balloting objections. The most obvious examples are the addition of reliable signal considerations to **Signals** §3.3 (including the addition of **Non-Local Jumps** §8.3.1) and the resolution of **Device- and Class-Specific Functions** §7. See also **Specific Derivations** §B.1.3.3.

Because there were too many notes interpolated in the text of the Trial Use Standard (which were nonetheless not part of the standard), and because there were still not enough to explain why the Working Group had made many difficult decisions, the Working Group decided to add a Rationale and Notes Appendix, modeled after the one the X3J11 Working Group was producing for the C Standard. Most of the notes formerly in the main body of the draft were moved to the Rationale appendix, although some were deleted and others were incorporated into the text of the standard proper.

B.1.5.2 Organization of this Appendix

Just as the standard proper excludes all examples, footnotes, references, and appendices, this Rationale is also not part of the standard. The POSIX interface is defined by the standard alone. If any part of this Rationale is not in accord with that definition, the IEEE Standards Office should be so informed. In the meantime, conflicts between this Rationale and the standard are always resolved in favor of the body of the standard.

All sections of this appendix after this first major section, **Introduction** §B.1, follow the exact structure of the standard, and aspects of a given section of the standard are considered in the corresponding section of the Rationale. Where a given discussion touches on several areas, attempts have been made to include cross-references within the text.

References to the standard are in the same format as references within the standard to parts of itself, for example: **Definitions** §2.0. References to this Rationale are given as references to Appendix B of the standard, that is, the section numbers always begin with "B." as in **Definitions** §B.2.0. Where a reference both to part of the standard and to a related note in the Rationale would be appropriate only the latter is given, because all parts of the Rationale implicitly refer to the corresponding parts of the standard.

B.1.5.3 Typographical Conventions

Words in all capital letters (including error numbers, environment variables, and limits) are one point size smaller than regular text, e.g.: POSIX.

Reference	Example
Command Name	`cpio`
Data Types	*long*
Defined Terms	**file**
Environment Variables	**PATH**
Error Numbers	[EINTR]
Function Arguments	*arg0*
Functions	*open*()
Global Externals	*errno*
Header Files	**<sys/stat.h>**
Limits	{OPEN_MAX}
Section References	**Process Termination** §3.2
Symbolic Constants	{_POSIX_V_DISABLE}

Defined names that are normally in lowercase, particularly function names, are never used at the beginning of a sentence or anywhere else that normal English usage would require them to be capitalized.

The above typographical conventions apply to both the standard and to this Rationale. There are also some conventions peculiar to the Rationale, regarding standards for the operating system interface and for the C language. These are used frequently in C

Language, X3J11, and P1003.1 §B.1.4:

Topic	Operating System Interface	C Programming Language
Working Group	P1003.1	X3J11
standard	IEEE Std 1003.1	*ANSI/X3.159-198x Programming Language C Standard*
short name	POSIX	C Standard
Rationale	Appendix B	*Rationale for American National Standard for Information Systems—Programming Language C*
short name	this Rationale	X3J11 Rationale

The name POSIX is usually used for the IEEE Std 1003.1 instead of the name 1003.1, because the latter is too easily confused with the name of the Working Group, P1003.1.

"Standard C" will eventually come to mean "ISO C," but currently refers to the *ANSI/X3.159-198x Programming Language C Standard* produced by the X3J11 Working Group.

B.2 Definitions and General Requirements

B.2.1 Terminology

The meanings specified in the standard for the words "shall," "should," and "may" are mandated by IEEE.

In this Rationale, the words "shall," "should," and "may" are sometimes used to illustrate similar usages in the standard. However, the Rationale itself does not specify anything regarding implementations or applications; see Organization of this Appendix §B.1.5.2.

implementation defined

This definition is analogous to that of the 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.

shall

Declarative sentences are sometimes used in the standard as if they included the word "shall," and facilities thus specified are no less required.

should

In this standard, 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.

undefined

See **implementation defined.**

unspecified

See **implementation defined.**

B.2.2 Conformance

The definition of **conforming implementations** §2.2.1 allows application developers to know what they can 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 the standard 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 the present standard.

The definitions of a **Conforming Application Using Extensions** §B.2.2.2 and of a **Strictly Conforming Application** §B.2.2.3 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 two definitions are modeled after related ones in the C Standard.

These three conformance definitions are descended from those of **conforming implementation, conforming application,** and **conforming portable application,** respectively, of the Trial Use Standard, but were changed to clarify

1. extensions, options, and limits,
2. relations among the three terms, and
3. relations between POSIX and the C Standard.

B.2.2.1 Implementation Conformance

B.2.2.1.1 Requirements

The word "support" is used rather than "provide" in order to allow an implementation that has no resident software development facilities but which supports the execution of a **Strictly Conforming Application** to be a **conforming implementation.** See also **Translation vs. Execution Environment** §B.1.4.

B.2.2.1.2 Documentation

The conforming documentation should use the same numbering scheme as this standard for purposes of cross referencing. (This also eliminates the need for a definitive "laundry list.")

This proposal is consistent with and supplements the verification test suite developed by the P1003.3 working group. All options that an implementation chooses should be listed in **<limits.h>** and **<unistd.h>**.

Hardware Failures: Many systems incorporate buffering facilities, maintaining updated data in volatile storage and transferring such updates to nonvolatile 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 arbitrarily bad. For such a user on 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 this standard. 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.

B.2.2.2 Application Conformance

B.2.2.2.1 Strictly Conforming Application

This definition is analogous to that of a Standard C conforming program.

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

Due to possible requirement for configuration or implementation characteristics in excess of the specifications in **<limits.h>** §2.9 or related to the hardware (such as array size or file space), not every Conforming Application Using Extensions will run on every conforming implementation.

B.2.2.2.2 Conforming Application

B.2.2.2.3 Conforming Application Using Extensions

B.2.2.3 Language Conformance

B.2.2.3.1 C Language Binding

The information concerning the use of library functions was adapted from a description in the 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:

1. by use of its associated header (possibly generating a macro expansion)

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

2. 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);
```

3. by explicit declaration

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

4. by implicit declaration

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

(Assuming no function prototype is in scope. This is not allowed by X3J11 for functions with variable arguments; furthermore, parameter type conversion ''widening'' is subject to different rules in this case.)

Note that the 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 would 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 is 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.2.3 General Terms

Many of these definitions are necessarily circular, and some of the terms (such as **process**) are variants of basic computing science terms that are notoriously hard to define. Some are defined by context in the prose topic descriptions of **General Concepts** §2.4, but most appear in the alphabetical glossary format of **General Terms** §2.3. All technical terms not explicitly defined have definitions in the *IEEE Dictionary*. See **Bibliographic Notes** §B.11.1.

Some definitions must allow extension to cover terms or facilities that are not explicitly mentioned in the standard. For example, the definition of **file** must permit interpretation to include streams, as found in Version 8. The use of abstract intermediate terms (such as **object** in place or in addition to **file**) has mostly been avoided in favor of careful definition of more traditional terms.

Some terms in the following list of notes do not appear in the standard; these are marked with a prepended asterisk (*). Many of them have been specifically excluded from the standard 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 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, the standard allows more granular privilege mechanisms to be defined. For many existing implementations of the UNIX system, the presence of the term **appropriate privileges** in this standard may be understood as a synonym for **super-user** (UID 0). However, future systems will undoubtedly emerge where this is not the case and each discrete controllable action will have **appropriate privileges** associated with it.

controlling terminal

The question of which of possibly several special files referring to the terminal is meant is not addressed in the standard.

***cooperating implementation**

This refers to a POSIX implementation that is done in combination with some other set of system specifications. This might be as simple as supporting a POSIX environment concurrently with some specific version of AT&Ts UNIX Operating System, or as complex as providing the POSIX environment with some different vendor's products, such as MS/DOS from Microsoft, VMS from Digital Equipment Company, etc. A cooperating environment would fall somewhere on the gray scale from hosted implementations to native, depending on the degree of POSIX components that are serviced directly versus those that are converted to correspond with one of the other system's implementations. (Note that the POSIX facilities might be native, and the other system hosted; or both might be native.)

***device number**

The concept is handled in *stat*() §5.6.2 as **ID of device**.

directory

The format of the directory file is implementation defined, and differs radically between System V and 4.3BSD. However, routines (derived from 4.3BSD) for accessing directories are provided in **Directory Operations** §5.1.2 and certain constraints on the format of the information returned by those routines are made in **Format of Directory Entries** §5.1.1.

directory entry

Throughout the document, the term **link** is used (about *link*() §5.3.4, for example) in describing the things that point to files from directories.

dot

The symbolic name **dot** is carefully used in the standard to distinguish the working directory filename from period or decimal point.

dot-dot

Historical implementations permit the use of these filenames without their special meanings. Such use precludes any meaningful use of these filenames by a Conforming Application. Therefore such use is considered an extension, the use of which makes an implementation non-conforming. See also **pathname resolution** §B.2.4.

Epoch

Normally, the origin of UNIX system time is 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. The concept of leap seconds is added for precision; at the time this standard was published, 18 leap seconds had been added since January 1, 1970. These 18 seconds are ignored to provide an easy and compatible method of computing time differences.

FIFO special file

See **pipe** §B.2.3.

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 system

Historically the meaning of this term has been overloaded with two meanings: that of the complete **file hierarchy** §B.2.4, and that of a mountable subset of that hierarchy, i.e., a **mounted file system** §B.2.3. The standard uses the term **file system** in the second sense, except that it is limited to the scope of a process (and a process's root directory). This usage also clarifies the domain in which a file serial number is unique.

***group file**

Implementation defined; see Passwords §B.9.

***historical implementations**

This term is used only in this appendix, not in the standard. It refers to previously-existing implementations of programming interfaces and operating systems that are related to the interface specified by the standard, especially to those implementations described by the Base Documents §B.1.3. See also Minimal Changes to Historical Implementations §B.1.2.8.

***hosted implementation**

This refers to a POSIX implementation that is accomplished through interfaces from the POSIX 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 *mknode*().) There is no necessary relationship between the type of implementation and its correctness, performance, and/or reliability.

***implementation**

The 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 were allowed, this usage would not have occurred.

***incomplete path name**

Absolute pathname §2.4 has been adequately defined.

***kernel**

See system call.

***library routine**

See system call.

***logical device**

Implementation defined.

***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.

***native implementation**

This refers to an implementation of POSIX that interfaces directly to an operating system kernel addressed in the standard. See also **hosted implementation** §B.2.3 and **cooperating implementation** §B.2.3. A similar concept from the UNIX world is a native UNIX system, which would a be kernel derived from one of AT&T's UNIX products.

***passwd file**

Implementation defined; see Passwords §B.9.

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:

open file description

open instance, file access description, open file information, and file access information.

file descriptor

file handle, file number [c.f., *fileno*].

pipe

It proved convenient to define a **pipe** as a special case of a FIFO even though historically the latter were only introduced in System III and do not exist at all in 4.3BSD.

portable filename 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 the standard. See also **filename portability** §B.2.4.

regular file

The standard does not intend to preclude the addition of structuring data (e.g., record lengths) in the file, as long as such data is not visible to an application that uses the features described in the standard.

root directory

This definition permits the operation of *chroot*(), even though that function is not in the standard. See also **file hierarchy** §B.2.4.

***root file system**

Implementation defined.

***root of a file system**

Implementation defined. See **mount point**.

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 event.

***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 the standard. See **Interface, Not Implementation** §B.1.2.2.

***super-user**

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

B.2.4 General Concepts

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*() §5.6.3. An implementation may include other security mechanisms in addition to those specified in the standard, 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 optional supplementary group IDs provide another reason for a process to not attempt to anticipate the result of an access attempt.

file hierarchy

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

- As noted in the standard, links may join branches.
- In some network implementations, there may be no single absolute root directory. See **pathname resolution**.
- With symbolic links (found in 4.3BSD), the file system need not be a tree or even a Directed Acyclic Graph.

file permissions

Examples of implementation defined constraints that may deny access are mandatory labels and access control lists.

filename portability

Most historical implementations, including all of those described by the **Base Documents** §B.1.3, prohibit case folding in filenames, i.e., treating upper- and lowercase alphabetic characters as identical. However, some consider case folding desirable

1. For user convenience.
2. For ease of implementation of the standard interface as a hosted system on some popular operating systems, which is compatible with the goal of making the standard interface **broadly implementable** §B.1.2.7.

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:

1. No solid evidence has been produced as to whether case sensitivity or case insensitivity is more convenient for users.
2. Making case insensitivity a POSIX 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 implementors would still have other problems with native file systems, such as short or otherwise constrained filenames, not to mention the lack of hierarchical directory structure.
3. 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 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.

4. 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.
5. 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 filenames; 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.
6. The file system implementation historically deals only with bytes, not with characters, except for slash and the null byte.
7. The purpose of the Working Group is to standardize the common, existing definition §B.1.2.1 of the UNIX system programming interface, not to change it. Mandating case insensitivity would make all historical implementations non-standard.
8. Not only the interface, but also application programs would need to change, counter to the purpose of having minimal changes to existing application code §B.1.2.9.
9. 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 the standard 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 filename character set definition and filename 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 the standard, the wording should be stronger to discourage the practice.

The consensus of the Working Group was in favor of proposal 1. Otherwise, a portable application would have to assume that case folding would occur when it wasn't wanted, but that it wouldn't occur when it was wanted.

file times update

General Concepts §2.4 has been changed to follow historical implementations. The times are not updated immediately, but are only marked for update by the functions.

pathname resolution

What the filename **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 the standard. In some networked systems the construction **/../hostname/** is used to refer to the root directory of another host, and the standard permits this behavior.

Other networked systems use the construct **//hostname/** for the same purpose, i.e., a double initial slash is used. The Working Group decided to prohibit this practice, because if such a construction is not equivalent to a single leading slash, it is more difficult to write shell scripts that depend on concatenating a directory name with a filename part. The utility (and ubiquitousness) of such shell scripts was considered more important than a particular file system implementation. This consideration did not apply to **/../hostname**, because that construct would not be used unless the application was deliberately accessing the network facility.

The term **root directory** is only defined in the standard 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.

B.2.5 Error Numbers

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.

A successful function call may set the value of *errno* to zero, or to any other value (except where specifically prohibited: see *mkdir*() §B.5.4.1). But it is meaningless to do so, since the value of *errno* is undefined except when the description of a function explicitly states that it is set, and no function description states that it should be set on a successful call. Most functions in most implementations do not change *errno* on successful completion. Exceptions are *isatty*() §4.7.2 and *ptrace*(). The latter is not in the standard, but is widely implemented and clears *errno* when called.

The standard requires (in the **Errors** subsections 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 **Error Numbers** §2.5. This section otherwise allows wide latitude to the implementation in handling error reporting. All references to the term **system call** have been excised from the descriptions of errors in this section.

Following each one-word symbolic name for an error, there is a one-line tag, which is followed by a description of the error. The one-line tag is merely a mnemonic or historical referent and is not part of the specification of the error. Many programs print these tags on the standard error stream (often by using the Standard C *perror*() function) when the corresponding errors are detected, but the standard does not require this action.

[EFAULT] Most historical implementations do not catch an error and set *errno* when a bad address is given to the functions *wait*() §3.2.1, *time*() §4.5.1, or *times*() §4.5.2. Some implementations cannot reliably detect a bad address. And most systems that detect bad addresses will do so only for a **system call** §B.2.3, not for a **library routine** §B.2.3.

[EINTR] The standard does not prohibit implementations from restarting interrupted system calls, nor does it require that [EINTR] be returned when another legitimate value may be subsituted, e.g., a partial transfer count when *read*() or *write*() are interrupted.

[ENAMETOOLONG]

[ENOMEM] The term **main memory** §B.2.3 has been eliminated from this description as being implementation defined.

[ENOTTY] The symbolic name for this error is derived from a time when device control was done by *ioctl*() §B.2.5 and that operation was only permitted on a terminal interface.

B.2.6 Primitive System Data Types

In early drafts, the standard specified that additional types that the implementation could place into **<sys/types.h>** had to be named with a "_t" suffix. This restriction was removed as it did not aid application portability and many implementations already were in violation.

clock_t — Traditionally, the type *time_t* was used for this. The Trial Use Standard used *ttime_t*. The present type was adopted to match the C Standard. See **Time** §B.4.5.

dev_t — This type may be made large enough to accomodate host-locality considerations of networked systems.

This type must be integral. Earlier drafts allowed this to be non-integral and provided a *samefile*() function for comparison.

mode_t — This type was chosen so that implementations could choose the appropriate integral type, and for compatibility with the C Standard. 4.3BSD uses **unsigned short** and the *SVID* uses **ushort**, which is the same thing. Historically, only the low-order sixteen bits are significant.

nlink_t — This type was introduced in place of **short** for **st_nlink** §5.6.1 in response to an objection that **short** was too small.

off_t — This type is used only in *lseek*() §6.5.3 and **<sys/stat.h>** §5.6.1. Many implementations would have difficulties if it were defined as anything other than **long**. The Working Group realizes that requiring an integral type limits the capabilities of *lseek*() to four gigabytes. See *lread*() §B.6.4. Also, the C Standard supplies routines that use larger types: see *fgetpos*() §B.6.5.3 and *fsetpos*() §B.6.5.3.

pid_t — This type has been proposed, but was not approved by the Working Group, because **int** is in common use on known systems, and sufficient need for *pid_t* to justify cost of changes has not been demonstrated. Also, many applications assume the digital representation of a process ID has a maximum of five digits; thus a larger type would not be of much use without requiring change of all such applications.

uid_t — Before the addition of this type, the data types used to represent these values varied throughout the standard. The **<sys/stat.h>** §5.6.1 header defined these values as type **short**, the **<passwd.h>** file (now **<pwd.h>** §9.2.2 and **<grp.h>** §9.2.1) used an **int** and *getuid*() §4.2.1 returned an **int**. In response to a strong objection to the inconsistent definitions, the Working Group decided to switch all the types 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 main problem associated with this change is a concern about 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 §9.2 structure. Consequently, implementations with an overriding concern for object compatibility can pad the structure back to its current size. For that reason, this problem wasn't considered critical enough to warrant the addition of a separate type to the standard.

B.2.7 Environment Description

LC_*

LC_* acknowledges the fact that the interfaces presented in the draft are not complete and may be extended as new international functionality is required. In the ANSI X3J11 draft proposal, names preceded by "LC_" are reserved in the name space for future categories.

To avoid name clashes, new categories and environments variables will be divided into two classifications: implementation-independent and implementation-dependent.

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-dependent names must be in lower-case letter, as below:

LC_*name*

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 Korn shell conform to the standard on this point. The usual name of dot §2.3 may also be used to refer to the current working directory.

TZ See *setlocale* () §8.1.2 for an explanation of the format.

LOGNAME 4.3BSD uses the environment variable USER for this purpose. 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.

B.2.8 C Language Definitions

The construct **<name.h>** for headers is also taken from the C Standard.

B.2.9 Numerical Limits

This section has been completely rewritten since the Trial Use Standard, in order to clarify the scope and mutability of several classes of limits.

The standard does not require an application to include **<limits.h>** everywhere a limit in it is used because many of them are system or application compile time constants that are not useful at runtime.

If the **translation and execution environments** §B.1.4 are actually distinct, it may be difficult to obtain information about runtime limits in the execution environment, especially considering that the C Standard does not even require the limits of **<limits.h>** to be kept in a file (they could instead be built into the translator). A useful technique is to write a small application that does nothing when run but report back on relevant limits.

The language in the first paragraph about `#if` preprocessing directives is taken from the C Standard.

B.2.9.1 C Language Limits

See also C Language Definitions §2.8 and **C Language, X3J11, and P1003.1** §B.1.4.

{CHAR_MIN}
: It is possible to tell if the implementation supports native character comparison as signed or unsigned by comparing this limit to zero.

{WORD_BIT}
: This limit has been omitted, as it is not referenced elsewhere in POSIX.

No limits are given in **<limits.h>** for floating point values because none of the functions in the standard proper use floating point values and all the functions that do that are imported from the C Standard by **Referenced C Language Routines** §8.1 defined in the C Standard, as are the limits that apply to the floating point values associated with them.

Though limits to the addresses to system calls were proposed, it is not clear how to implement them for the range of systems being considered and, lacking a complete proposal, the Working Group determined not to attempt this at this time. Limits regarding hardware register characteristics were similarly proposed and not attempted.

B.2.9.2 Run-time Invariant Values

The criterion for inclusion of an item in this section is that a Conforming Application Using Extensions could break if the corresponding restriction is relaxed between the time the Conforming Application Using Extensions is translated and the time it is executed.

If, in a specific implementation, any of the parameters specified in this subsection can be varied at run time, the implementation will only be a conforming implementation when the values set at run time match those in the **<limits.h>** file.

The heading of the rightmost column of the table is given as "Minimum Value" rather than "Value" in order to emphasize that the numbers given in that column are minimal for the actual values a specific implementation is permitted to define in its **<limits.h>**. The values in the actual **<limits.h>** define, in turn, the maximum amount of a given resource that a Conforming Application can depend on finding when translated to execute on that implementation. A Conforming Application Using Extensions must function correctly even if the value given in **<limits.h>** is the minimum that is specified in the standard. (The application may still be written so that it performs more efficiently when a larger value is found in **<limits.h>**.) A conforming implementation must provide at least as much of a particular resource as that given by the value in the standard. An implementation that cannot meet this requirement (a "toy implementation") cannot be a conforming implementation.

{FCHR_MAX}

is specifically a measure of the addressability of bytes in a file. It was dropped from the standard in Draft 12. The value given implies that *off_t* must be at least 24 bits wide. In terms of testability, it should be possible to do the following on a conforming implementation:

Create a file with:

```
int file;
file = open(path, O_RDWR|O_CREAT|O_TRUNC, 0600);
lseek(file, (off_t)16777215, SEEK_SET);
write(file, '1', 1);
lseek(file, (off_t)0, SEEK_SET);
/* read 16777215 bytes with value 0 */
/* read 1 byte with value 1 */
```

There is *no* requirement that a conforming implementation provides the ability to create a non-sparse file containing 16777216 bytes (or any other number of bytes). It is expected, however, that it will be possible to configure specific instances of most specific implementations such that files of any required length less than or equal to {FCHR_MAX} + 1 can be created. A Conforming Application Using Extensions will generally depend on the ability to create non-sparse files of some specific length. It is the responsibility of the administrator who configures a specific instance of a specific implementation to provide adequate file storage space to allow applications to run. To put this another way, even a Conforming Application Using Extensions will not run on a specific instance of a specific implementation if less file storage space is provided than is required by the Conforming Application Using Extensions. The standard says nothing about available file space, just as it says nothing about available memory space.

{MAX_INPUT}

Since the only use of this limit is in relation to terminal input queues, it mentions them specifically. This limit was originally named {MAX_CHAR} in early drafts. Application writers should use {MAX_INPUT} primarily as an indication of the number of characters that can be written as a single unit by one Conforming Application Using Extensions communicating with another via a terminal device. It is *not* implied that input lines received from terminal devices always contain {MAX_INPUT} characters or fewer: an application that attempts to read more than {MAX_INPUT} characters from a terminal may receive more than {MAX_INPUT} characters.

{PATH_MAX}

A Conforming Application or Conforming Application Using Extensions that, for example, compiles to use different algorithms depending on the value of {PATH_MAX} should use code such as

```
#if defined(PATH_MAX) && PATH_MAX < 512
          ...
#else
#if defined(PATH_MAX) /* PATH_MAX >= 512 */
          ...
#else /* PATH_MAX indeterminate */
          ...
#endif
#endif
```

This is because the value tends to be very large or indeterminate on most historical implementations (it is arbitrarily large on System V). On such systems there is no way to quantify the limit, and it seems counter-productive to include an artificially small fixed value in <limits.h> in such cases.

B.2.9.3 Run-time Invariant Values (Possibly Indeterminate)

B.2.9.4 Pathname Variable Values

B.2.9.5 Run-time Increasable Values

Values appear in this section if there is no possibility that arbitrarily increasing them between the translation and the execution of a Conforming Application Using Extensions could break the Conforming Application Using Extensions. Specific instances of specific implementations may choose to increase the values in order to support non-portable applications.

Use of the word "may" in "...may increase the value" is correct. P1003.3 need not test whether the value is less restrictive than that given in **<limits.h>** or by how much.

A {DIR_LEVEL_MAX} limit was removed from the draft because it had no perceived value to an application.

B.2.9.6 Bounded Ranges of Values

A Conforming Application can assume that it can have *at least* the most restrictive value of the resource. It has a "fighting chance" (a phrase used by P.J. Plauger of X3J11) of getting as much as that given by the least restrictive value. It can *never* get more than that given by the least restrictive value. The utility of the bounded range concept is that it allows the following:

a) If a Conforming Application wants (for example) to close all open files, the least restrictive value tells it how many close operations are needed in order to ensure that all files have been closed. Without knowledge of the value, this number is indeterminate.

b) The intention is that a supplier of a range of compatible computers should be able to ship a single **<limits.h>** which adequately describes the entire range. Thus if, for example, **<limits.h>** for a superminicomputer contains the pair

```
#define   OPEN_MAX        20
#define   OPEN_MAX_CEIL   80
```

an application running on the same vendor's workstation is entitled to expect that it can have 20 open files (and may legitimately malfunction if it is not able to do so). The same binary application code, when running on a much larger member of the same machine family may find that it can have as many as 80 open files. An intelligently-written application may be able to optimize its algorithms according to the amount of a particular resource that it can obtain, but should not attempt to obtain more of any resource than that indicated by the corresponding upper limit defined by **<limits.h>**.

> Looking at the same issue from another angle, the vendor need only ship one C compiler package for the entire machine family; an application developer need only compile once to produce a program that runs optimally across the entire range of machines in the family.

Use of the word "may" in "may relax the corresponding restriction" is correct, but raises a testability issue. If, for example, **<limits.h>** suggests that it *may* be possible for a process to open as many as 80 files, but never to be able to open the eighty-first, P1003.3 must insist that this condition can be attained.

{CHILD_MAX}
: In a typical implementation, one process per user ID is used for the login shell, and one for the current process, leaving four potential children.

{LOCK_MAX}

{PROC_MAX}

{SYS_OPEN}
: These three limits were removed from **<limits.h>**. The information in **<limits.h>** should be useful to a Conforming Application; these three values were not useful: it is of no use, for example, for a Conforming Application to know the size of the system open file table, as there is no way that a process group, for instance, can ever be sure how many of those files it can open. The only thing that is certain is that each process in the group may be able to open no more than {OPEN_MAX} files, and may be able to open as many as {OPEN_MAX_CEIL}. **<limits.h>** implies this. {SYS_OPEN} does not add to the useful information available to the Conforming Application.

B.2.10 Symbolic Constants

B.2.10.1 Symbolic constants for the *access*() function

B.2.10.2 Symbolic constants for the *lseek*() function

B.2.10.3 Symbolic constants for portability specifications

B.2.10.4 Compiler time symbolic constants for portability specifications

Related material appeared in an appendix of the Trial Use Standard. The purpose there was to allow an application developer to have a chance to determine whether a given application would run (or run well) on a given implementation. To this purpose has been added that of simplifying development of verification suites (see Verification Testing §A.2.3) for the standard. The constants given here were originally proposed for a separate file, **<posix.h>**, but the Working Group decided that they should appear in **<unistd.h>** along with other symbolic constants.

B.2.10.5 Execution time symbolic constants for portability specifications

Without the addition of {_POSIX_NO_TRUNC} and _PC_NO_TRUNC to the Configurable Open Variables list, the Standard says nothing about the effect of a pathname component longer than {NAME_MAX}. There are only two effects in common use in implementations: truncation, or an error. It is desirable to limit allowable behavior to these two cases. It is also desirable to permit applications to determine what an implementation's behavior is, because services that are available with one behavior may be impractical to provide with the other. However, since the behavior may vary from one file system to another, it may be necessary to use *pathconf*() to resolve it.

B.3 Process Primitives

B.3.1 Process Creation

A common way to produce ("spawn") a descendant process that does not need to be waited on is to *fork*() to produce a child and *wait*() on the child. The child *fork*()s again to produce a grandchild. The child then exits and the parent's *wait*() returns. The grandchild is thus disinherited by its grandparent.

A simpler method (from the programmer's point of view) of spawning is to do

```
system("something &");
```

However, this depends on features of a process (the shell) that are outside the scope of the present standard, although they may be addressed by P1003.2.

B.3.1.1 Process Creation

During the *fork*() function call, signals directed to a group of processes, of which the child process is a member, may fail to be delivered to the child process. See *kill*() §B.3.3.2.

Many existing implementations have timing windows where a signal sent to a process group (e.g. an interactive SIGINT) just prior to or during execution of *fork*() is delivered to the parent following the *fork*() but not the child, because the *fork*() code clears the child's set of pending signals. It is not the intention of this standard to require, or even permit, this behavior. This behavior is only a consequence of the implementation failing to make the interval between signal generation and delivery totally invisible. From the application's perspective, a *fork*() call should appear atomic. A signal that is generated prior to the *fork*() should be delivered prior to the *fork*(). A signal sent to the process group after the *fork*() should be delivered to both parent and child. The implementation might actually initialize internal data structures corresponding to the child's set of pending signals to include signals sent to the process group during the *fork*(). Since the *fork*() call can be considered as atomic from the application's perspective, from that view the set would be initialized as empty and such signals would have arrived after the *fork*(). See also pending signals §B.3.3.6.

The [EINTR] error was considered too implementation-specific to include.

B.3.1.2 Execute a File

The value of *argc*, and the corresponding number of non-null *argv* pointers, should be adjusted by the implementation so that *main*() receives at least one argument even when the *exec*() call that invoked it supplied none. This is both because existing programs expect it and also in order to conform with the C Standard.

A **Strictly Conforming Application** §2.2.3 is required to supply an *arg0* that points to a filename associated with the new process image file, and a **Conforming Implementation** §2.2.1 is required to supply such an argument to *main*() in *argv*[0] (even if the calling application did not). But no such requirement is placed on **Application Conformance** §2.2.2, due to the use of the word "should" rather than "shall."

Some implementations provide a third argument to *main*() called *envp*. This is defined as a pointer to the environment. The C Standard provides *environ*, which replaces all need for the *envp* argument. Implementations are required to support the two-argument calling sequence, but this does not prohibit an implementation from supporting *envp* as an optional, third argument.

If the saved set-user-ID/saved set-group-ID option is implemented, *exec*() always saves the *uid* and *gid* of the process prior to the *exec*().

[E2BIG] The limit {ARG_MAX} applies not just to the size of the argument list, but to the sum of that and the size of the environment list.

[EFAULT]

Some existing systems return [EFAULT] rather than [ENOEXEC] when the new process image file is corrupted. They are non-conforming.

[ETXTBSY] The error [ETXTBSY] was considered too implementation-dependent to include. System V returns this error when the executable file is currently open for writing by some process. The standard neither requires nor prohibits this behavior.

B.3.2 Process Termination

"Abnormal termination with actions" includes, in most historical implementations, the creation of a file named `core` in the current working directory of the process. This file contains an image of the memory of the process, together with descriptive information about the process, perhaps sufficient to reconstruct the state of the process at the receipt of the signal.

There is a potential security problem in creating a `core` file if the process was set-user-ID and the current user is not the owner of the program, if the process was set-group-ID and none of the user's groups match the group of the program, or if the user does not have permission to write in the current directory. In this situation, an implementation either should not create a **core** file or should make it unreadable by the user.

The name of the file is not mentioned in the standard because some historical implementations use a different name, such as by appending the process ID to the filename. However, applications are advised not to create files named **core** because of potential conflicts in many implementations.

B.3.2.1 Wait for Process Termination

See *_exit*() §B.3.2.2.

The status values are given as specific bit encodings because they are that way in most historical implementations and many existing programs expect it.

A call on the *wait*() function only returns status on an immediate child process of the calling process, i.e., a child that was produced by a single *fork*() §3.1.1 call (perhaps followed by an *exec* §3.1.2 or other function calls) from the parent. If a child produces grandchildren by further use of *fork*(), none of those grandchildren nor any of their descendants will affect the behavior of a *wait*() from the original parent process.

The *wait2*() function is provided for **job control** §B.3.3. It is identical to the *wait3*() function provided by 4.3BSD except that the third argument, the returned resource usage summary, is not provided since it is not directly relevant to job control. The *wait2*() function can be implemented as a library function on top of *wait3*().

Appendix E provides an alternative proposal for the *wait* family. Currently, there is no way to write a library routine, such as *system*() or *pclose*(), without interfering with other zombies. For example, consider the problem that which the P1003.2 group addressed:

```
stream = popen("/bin/true");
(void) system("sleep 100");
(void) pclose(stream);
```

On all systems since Version 6, the final *pclose*() will fail to reap the wait status of the *popen*().

This proposal changes section 3.2.1 by augmenting the *wait2*() call in several ways:

wait2() has been given a more descriptive name of *waitpid*().

waitpid() can wait for a specific child, a child in the current process group, or a child in a specific process group. The use of *pid* corresponds to the use of *pid* in *kill*().

waitpid() is required, and the WUNTRACED related actions are defined only for systems that have the Job Control Option.

It should be noted that:

```
waitpid(stat_loc, -1, options)
```

provides the same functionality as the function in the body of the standard:

```
wait2(stat_loc, options)
```

The *waitpid*() function solves some major problems related to the functions *system*(), *popen*(), and *pclose*() for Version 6, Version 7, Version 8, Version 9, System III, System V, and 4BSD-based systems.

The *waitpid*() function would also greatly help in the writing of portable command interpretors.

B.3.2.2 Terminate a Process

The function *_exit*() is defined here instead of *exit*() because the C Standard defines the latter to have certain characteristics that are beyond the scope of the present standard, specifically the flushing of buffers on open files and the use of *atexit*(). See **C Language and X3J11** §B.1.5. There are several public domain implementations of *atexit*() which may be of use to interface implementors who wish to incorporate it.

It is important that the consequences of process termination as described in this section occur regardless of whether the process called *_exit*() (perhaps indirectly through *exit*()) or instead was terminated due to a signal or for some other reason. See also **Process Termination** §B.3.2.

A language other than C may have other termination primitives than the C language *exit*() function, and programs written in such a language should use its native termination primitives, but those should have as part of their function the behavior of *_exit*() as described in this section. Implementations in languages other than C are outside the scope of the present standard, however.

As required by X3J11, using *return*() from *main*() §3.1.2 is equivalent to calling *exit*() with the same argument value. Also, reaching the end of the *main*() function is equivalent to using *exit*() with an unspecified value.

Historically, the implementation-dependent process that inherits children whose parents have terminated without waiting on them is called **init**, and has process ID 1.

The distinction between session process group leaders and job control process group leaders was created to allow the 4.2BSD semantics necessary to support job control without precluding the semantics of System V. System V sends the SIGHUP signal to the process group of a terminating process group leader. Such a process group leader is typically a login shell. 4.2BSD does not send SIGHUP under these conditions for two reasons:

- First, job control semantics preclude killing background jobs at logout. While System V provides the `nohup` command to prevent killing background processes at logout, the user must make the decision when launching the command. The point of job control is that such decisions can be changed after launching the command.

- Second, every command pipeline launched by a job control shell (such as `csh`) resides in its own unique process group with one command in the pipeline being the process group leader. If SIGHUP were sent to the process group when that process terminated, the remaining pipeline would be prematurely terminated.

If the terminating process has any children which are currently stopped, those children will be sent SIGHUP immediately followed by SIGCONT. This continues the stopped children and, unless they are catching or ignoring SIGHUP, also causes them to terminate. The goal is to prevent stopped processes from languishing forever. When a process exits with stopped children, those children are no longer under the control of a job control shell and hence would not normally ever be continued. See also the discussion of sending SIGKILL to stopped orphaned processes in **Signal Names** §B.3.3.1.

B.3.3 Signals

Signals, as defined in the Trial Use Standard, and in Version 7, System III, the *1984 /usr/group Standard*, and System V (except very recent releases), have shortcomings which make them unreliable for many application uses. Several objections have been voiced to the Trial Use Standard because of this. Therefore a new signal mechanism, based very closely on the one of 4.2BSD and 4.3BSD, was added to the standard. With the exception of two features (see item 4 below and also **Examine Pending Signals** §B.3.3.6), it is possible to implement the POSIX interface as a simple library veneer on top of 4.3BSD.

The major differences from the BSD mechanism are:

1. **Signal mask type.**
   BSD uses the type **int** to represent a signal mask, thus limiting the number of signals to the number of bits in an **int** (typically thirty-two). The new standard instead uses a defined type for signal masks. Because of this change, the interface is significantly different than in BSD implementations, although the functionality and potentially the implementation are very similar.

2. **Restarting system calls.**
   Unlike all previous historical implementations, 4.2BSD restarts some interrupted system calls rather than returning an error with *errno* set to [EINTR] after the signal-catching function returns. This change caused problems for some existing application code. 4.3BSD and other systems derived from 4.2BSD allow the application to choose whether system calls are to be restarted. The standard (in *sigaction*() §3.3.4) does not require restart of functions, because it was not clear that the semantics of system call restart in any existing implementation were useful enough to be of value in a standard. Implementors are free to add such mechanisms as extensions.

3. **Signal stacks.**
   The 4.2BSD mechanism includes a function *sigstack*(). The 4.3BSD mechanism includes this and a function *sigreturn*(). No equivalent is included in the standard because these functions are not clearly portable or necessary. See also **Non-local Jumps** §8.4.

4. **Pending signals.**
   The *sigpending*() §3.3.6 function is the sole new signal operation introduced in the standard. It was requested by some members of the Working Group and was seen as a simple and useful feature.

The Working Group considered making reliable signals optional. However, the consensus was that this would hurt application portability, as a large percentage of applications using signals can be hurt by the unreliable aspects of the *signal*() §B.8.3.2 mechanism. This unreliability stems from the specification that the signal action is reset to SIG_DFL before the user's signal-catching routine is entered.

Most traditional implementations do not queue signals, i.e., a process's signal handler is invoked once, even if the signal has been generated multiple times before it is delivered. A notable exception to this is SIGCLD which, in System V, is queued. The Working Group decided to neither require nor prohibit the queueing of signals. It is expected that a future Real Time Extension to this standard (see **Real Time Extensions** §A.2.4) will address the issue of reliable queueing of event notification.

Note that an application which simply catches the interactive SIGINT signal with *signal*() can be terminated with no chance to recover when two such signals arrive sufficiently close in time (e.g., when a user gets impatient on a busy system).

Job Control.

The intent in adding 4.2BSD-style **job control** functionality was to adopt the necessary 4.2BSD 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 Applications.

Discussions of the changes can be found in the sections which discuss the specific interfaces. See sections: **Wait for Process Termination** §B.3.2.1, **Terminate a Process** §B.3.2.2, **Signal Names** §B.3.3.1, **Send a Signal to a Process** §B.3.3.2, **Examine and Change Signal Action** §B.3.3.4, **Set Process Group** §B.4.3.2, **Job Access Control** §B.7.1.1.5, and **Set Distinguished Process Group ID** §B.7.2.4.

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 which continues the stopped process. Because a job control shell is allowed, but not required, by the standard, an implementation must provide a mechanism which shields processes from job control signals when there is no job control shell. The usual method is for the system initialization process (typically called **init**), which is the ancestor of all processes, to launch its children with the signal handling action set to SIG_IGN for the signals SIGTSTP, SIGTTIN, and SIGTTOU. Thus all login shells start with these signals ignored. If the shell is not job control cognizant, then it should not alter this setting and all its descendants should inherit the same ignored settings. At the point where a job control shell is launched, it resets the signal handling action for these signals to be SIG_DFL for its children and (by inheritance) their descendants. Also, shells which are not job control cognizant will not alter the process group of their descendants or of their controlling terminal; this has the effect of making all processes be in the foreground (assuming the shell is in the foreground).

POSIX does not specify how controlling terminal access is affected by a user logging out (that is, by a login shell terminating). 4.2BSD uses the *vhangup*() function to prevent any access to the controlling terminal through file descriptors opened prior to logout. System V does nothing to 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 Application should not attempt to access the controlling terminal after logout since such access is unreliable.

B.3.3.1 Signal Names

B.3.3.1.1 Synopsis

B.3.3.1.2 Description

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 the document 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 this document. An example of this is SIGABRT, which traditionally overlaps some other signal, such as SIGIOT.

SIGKILL, SIGTRAP, 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 the standard because their behavior is implementation dependent 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 SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, and SIGCONT are provided for job control and are unchanged from 4.2BSD. The signal SIGCLD is also typically used by job control shells to detect children which have terminated or, as in 4.2BSD, stopped. However, the 4.2BSD name, SIGCHLD, was dropped in favor of the System V SIGCLD. See also SA_CLDSTOP §B.3.3.4.

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*() §3.3.2. 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.

In actual existing implementations, there are a few cases where the interval between generation and delivery of unmasked signals is visible to applications. For example, a pending signal (masked or unmasked) is discarded when its signal action is set to SIG_IGN. Implementations should make this interval invisible to the extent possible. When this is totally true, references to pending signals apply only to pending, masked signals.

There is one case where a blocked signal does not remain pending until it is unblocked. In the System V implementation of *signal*(), there are some cases in which pending signals are also discarded when the action is set to SIG_DFL or a signal-catching routine.

In 4.2BSD and 4.3BSD, there is one other case where a blocked signal is not kept pending. When the signal is being ignored and is also blocked, it is discarded immediately on generation. The Working Group did not wish to standardize this behavior. Implementations which do this do not conform completely to this standard.

There is very little if anything that a Conforming 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 (e.g., 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's termination (with *wait*()) and print an appropriate message. The mnemonic tags for these signals are derived from their PDP-11 origin.

B.3.3.1.3 Signal Actions

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 the standard. The value of such a mechanism, if included, would be diminished given that SIGKILL would still not be catchable.

The specification of the effects of SIG_IGN on SIGCLD as implementation defined permits but does not require the System V effect of causing terminating children to be ignored by *wait*() §3.2.1. Yet it permits SIGCLD to be effectively ignored in an implementation-independent 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 the standard requires that the default action for SIGCLD be to ignore the signal, applications should always set the action to SIG_DFL in order to avoid SIGCLD.

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 4.3BSD, do not generate a new SIGCLD signal in this way. In general, a process should not attempt to alter the signal action for the SIGCLD signal while it has any outstanding children.

SIGCONT has no effect on a running process if the action is set to SIG_DFL, even though the signal will still cause a stopped process to continue.

If a process is orphaned (because its parent has terminated) and then subsequently 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 which stop are sent the SIGKILL signal which causes them to terminate. The goal is to prevent stopped processes from languishing forever. See also SIGCONT §B.3.3.1.

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. The standard does not address the more general problem of synchronizing access to shared data structures. Naturally, the same principles apply to the reentrancy of application routines and asynchronous data access. Note that *longjmp*() is not in the list of reentrant functions; applications that *longjmp*() out of signal handlers require rigorous protection in order to be portable.

B.3.3.2 Send a Signal to a Process

The semantics for permission checking for *kill*() differ between System V and most other implementations, such as Version 7 or 4.3BSD. The semantics chosen for the standard agree with System V. Specifically, a setuid process cannot protect itself against signals (or at least not against SIGKILL) unless it changes its real user ID. This choice allows the user who starts an application to send it signals even if it changes its effective user ID. The other semantics give more power to an application that wants to protect itself from the user who ran it.

The implementation defined processes to which a signal cannot be sent may include the scheduler or **init**.

As in 4.2BSD, the SIGCONT signal can be sent to any descendant process regardless of user ID security checks. This allows a job control shell to continue a job even if processes in the job have altered their user IDs (as in the **su** command). Note that this applies to all descendant processes, not just immediate children. A similar relaxation of security is not necessary for the other job control signals since those signals are typically sent by the terminal driver in recognition of special characters being typed; the terminal driver bypasses all security checks.

In secure implementations, a process may be restricted from sending a signal to a process having a different security label. In order to prevent the existence or non-existence of a process from being used as a covert channel, such processes should appear non-existent to the sender; i.e., [ESRCH] should be returned, rather than [EPERM], if *pid* refers only to such processes.

B.3.3.3 Manipulate Signal Sets

The implementation of the *siginitset*() function may reasonably be a no-op. It is also reasonable for it to initialize part of the structure, such as a version field, to permit binary compatibility between releases where the size of the set varies. This function is not intended for dynamic allocation.

B.3.3.4 Examine and Change Signal Action

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

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

The ellipsis (", ...") is Standard C 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 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 to be portable. This is in conflict with the goal of minimal changes to existing application code §B.1.2.9.

This problem with variable parameter lists does not apply to *open*(), *execl*(), *printf*(), and other functions written by implementor of Standard C or POSIX. The application developer does not have to provide a function parameter type definition of these functions, and the declaration used by the implementor of the standard will determine the mechanism used for passing variable argument lists.

The problem would also not occur for new facilities, since application writers could use the appropriate function parameter definition in their new code.

The Working Group has nonetheless chosen to avoid the use of variable argument syntax and of function parameter types in general in order to ease bindings of POSIX to languages other than Standard C. See **Conformance** §B.2.2 and **Function parameter type lists** §B.1.4.

The SA_CLDSTOP flag, when supplied in the *sa_flags* parameter, allows overloading SIGCLD with the 4.2BSD SIGCHLD semantics necessary for job control.

B.3.3.5 Examine and Change Blocked Signals

B.3.3.6 Examine Pending Signals

B.3.3.7 Wait for a Signal

Normally, at the beginning of a critical code section, a specified set of signals is blocked using the *sigprocmask*() function. When the process has completed the critical section and needs to wait for the previously blocked signal(s), it pauses by calling *sigsuspend*() with the mask that was returned by the *sigprocmask*() call.

B.3.4 Timer Operations

B.3.4.1 Process Alarm Clock

Because many traditional implementations (including Version 7 and System V) do allow an alarm to occur up to a second early, the Working Group did not feel it could disallow this behavior, and thus a Conforming Application needs to be prepared for it. However, the Working Group does not want to encourage this behavior. Other implementations allow alarms up to half a second early, up to 1/{CLK_TCK} seconds early, or do not allow them to occur early at all. The latter is considered most appropriate. Future real-time standards related to this one (see **Real Time Extensions** §A.2.4) may specify such facilities.

B.3.4.2 Suspend Process Execution

Many common uses of *pause*() have timing windows. The scenario involves checking a condition related to a signal and, if the signal has not occurred, calling *pause*(). When the signal occurs between the check and the call to *pause*(), the process often blocks indefinitely. The *sigprocmask*() and *sigsuspend*() functions can be used to avoid this type of problem.

B.3.4.3 Delay Process Execution

Traditional implementations often implement *sleep*() using *alarm*() and *pause*(). One such implementation is prone to infinite hangs as described in *pause*() §B.3.4.2. Another such implementation uses the C language *setjmp*() and *longjmp*() functions to avoid that window. That implementation introduces a different problem; when the alarm signal interrupts a signal catching function installed by the user to catch a different signal the *longjmp*() aborts that signal-catching function. An implementation based on *sigprocmask*(), *alarm*(), and *sigsuspend*() can avoid these problems.

Scheduling delays may cause the process to return from the *sleep*() function significantly after the requested time. In such cases, the return value should be set to zero, since the formula (requested time minus the time actually spent) yields a negative number and *sleep*() returns an **unsigned int**.

B.4 Process Environment

B.4.1 Process Identification

B.4.1.1 Get Process and Parent Process IDs

B.4.2 User Identification

B.4.2.1 Get Real User, Effective User, Real Group, and Effective Group IDs

B.4.2.2 Set User and Group IDs

Another way of looking at the behavior of these two functions:

> The call *setuid*(uid) shall result in both the real user ID and the effective user ID being equal to *uid* if:
>
> > the effective user ID is super-user
> > *or*
> > the real user ID is *uid*
> > *or*
> > the effective user ID is *uid* (implementation permitting).

The call *setgid*(gid) shall result in both the real group ID and the effective user ID being equal to *gid* if:

> the effective user ID is super-user
> *or*
> the real group ID is *gid*
> *or*
> the effective group ID is *gid* (implementation permitting).

The call *setuid*(uid) sets the effective user ID of the calling process to *uid* if any of the following conditions are met:

> The real user ID of the calling process is *uid*.
>
> The implementation provides the saved set-user-ID option and the saved set-user-ID for the calling process is *uid*.
>
> The process has appropriate privileges. In this case, the real user ID and optional saved set-user-ID are also set to *uid*.

The saved set-user-ID capability allows a program to regain the effective user ID established at the last *exec* §3.1.2 call. Similarly, the saved set-group-ID capability allows a program to regain the effective group ID established at the last *exec* call.

These last two capabilities are derived from System V. Without them, a program may have to run as super-user in order to perform the same functions, because super-user can write on the user's files. This is a problem because such a program can write on *any* user's files, and so must be carefully written to emulate the permissions of the calling process properly.

The ability to set the real user ID to the value of its effective user ID corresponds to the behavior of 4.2BSD and 4.3BSD. This is not a security risk over systems that do not implement it; it actually reduces the access capabilities of a process.

B.4.2.3 Get Supplementary Group IDs

The related function *setgroups*() §B.4.2.3 is a privileged operation and therefore is not covered by this standard.

The wording regarding the group of a newly created regular file, directory, or FIFO in *open*() §5.3.1, *mkdir*() §5.4.1, *mkfifo*() §5.4.2, respectively, uses "may" rather than "shall" in order to permit both the System V (and Version 7) behavior (in which the group of the new object is set to the effective group ID of the creating process) and the 4.3BSD behavior (in which the new object has the group of its parent directory). An application that needs a file to be created in the group of the effective group ID should use *chown*() §5.6.5 to ensure the new group regardless of the style of groups the interface implements.

B.4.2.4 Get User Name

L_cuserid must be defined appropriately for a given implementation and must be greater than zero so that array declarations using it are accepted by the compiler. The value includes the terminating null byte.

B.4.3 Process Groups

B.4.3.1 Get Process Group ID

4.3BSD provides a *getpgrp*() function that returns the process group ID for a specified process. Although this function is used to support job control, all known job control shells always specify the calling process with this function. Thus the simpler System V *getpgrp*() suffices and the added complexity of the 4.3BSD *getpgrp*() has been omitted from the standard.

B.4.3.2 Set Process Group ID

.br

B.4.3.3 Set Process Group ID for Job Control

The *jcsetpgrp*() function is similar to the *setpgrp*() function of 4.2BSD. The differences are:

4.2BSD *setpgrp*() allows the caller to specify the process ID of the process to affect. Since all known job control shells always affect the calling process, this parameter was deleted; the affected process is now always the calling process.

4.2BSD allowed the specified new process group to assume any value. This presents certain security problems and is more flexible than necessary to support job control. In keeping with the new security model (see Job Control §B.3.3), *jcsetpgrp*() only allows the calling process to join a process group that is already associated with the calling process' controlling terminal. One special case is where the calling process is creating a new process group, that is where there are no other processes currently in the process group. In this case, the calling process is allowed to join the new group.

These restrictions maintain the assertion that the calling process is not introducing a new (different) controlling terminal into an already existing process group. Violating this assertion would result in one process group (or job) which could be controlled by more than one controlling terminal (or login session). The typical scenario that is being prevented is for a process to first use *jcsetpgrp*() to join the process group of another login session and then to use *tcsetpgrp*() §7.2.4 to allow keyboard signals from its controlling terminal to affect processes in a different session.

One non-obvious use of *jcsetpgrp*() is to allow a job control shell to return itself to its original process group (the one in effect when the job control shell was executed). A job control shell does this before returning control back to its parent when it is terminating or suspending itself as a way of restoring its job control "state" back to what its parent would expect. (Note that the original process group of the job control shell typically matches the process group of its parent, but this is not necessarily always the case.) See also *tcsetpgrp*() §B.7.1.7.

B.4.4 System Identification

B.4.4.1 System Name

The values of the structure members are not constrained to have any relation to the version of this interface standard implemented in the operating system. An application implementor should instead depend on {_POSIX_VERSION} and related constants defined in Symbolic Constants §2.10.

The standard does not define the sizes of the members of the structure and permits them to be of different sizes, although most implementations define them all to be the same size: eight bytes plus one byte for the string terminator. That size for *nodename* is not enough for use with many networks.

The *uname*() function is specific to System III, System V, and related implementations, and it does not exist in Version 7 or 4.3BSD. The values it returns are set at system compile time in those existing implementations.

4.3BSD has *gethostname*() and *gethostid*(), which return a symbolic name and a numeric value, respectively. There are related *sethostname*() and *sethostid*() functions that are used to set the values the other two functions return. The length of the host name is limited to 31 characters in most implementations and the host ID is a thirty-two bit integer.

B.4.5 Time

The *time*() §4.5.1 function returns a value in seconds (type *time_t*) while *times*() §4.5.2 returns a set of values in {CLK_TCK}ths of a second (type *clock_t*).

Some historical implementations, such as 4.3BSD, have mechanisms capable of returning more precise times (see *gettimeofday*() §B.4.5.1). A generalized timing scheme to unify these various timing mechanisms has been proposed but not adopted in this standard; see Real Time Extensions §A.2.4.

B.4.5.1 Get System Time

Implementations in which *time_t* is a thirty two bit signed integer (most historical implementations) will fail in the year 2038. The Working Group chose not to try to fix this. But they did require the use of *time_t* in order to ease the eventual fix.

Many historical implementations (including Version 7) and the *1984 /usr/group Standard* use long instead of *time_t*. The present standard uses the latter type in order to agree with the C Standard.

4.3BSD includes *time*() only as an interface to the more flexible *gettimeofday*() §B.4.5.1 function.

B.4.5.2 Process Times

The inclusion of times of child processes is recursive, so that a parent process may collect the total times of all of its descendants. But the times of a child are only added to those of its parent when its parent successfully waits on the child. Thus it is not guaranteed that a parent process will always be able to see the total times of all its descendants.

If the type *clock_t* is defined to be a signed thirty-two bit integer, it will overflow in somewhat more than a year if {CLK_TCK} is 60, or less than a year if it is 100. There are individual systems that run continuously for longer than that. The standard permits an implementation to make the reference point for the returned value be the startup time of the process, rather than system startup time.

B.4.6 Environment Variables

B.4.6.1 Environment Access

Additional functions *putenv*() and *clearenv*() were considered but rejected because they were more oriented towards system administration than ordinary application programs.

B.4.7 Terminal Identification

The difference between *ctermid*() and *ttyname*() is that *ttyname*() must be passed a file descriptor and returns the pathname of the terminal associated with that file descriptor, while *ctermid*() returns a string (such as /dev/tty) that will refer to the controlling terminal if used as a pathname. Thus *ttyname*() is useful only if the process already has at least one file open to a terminal.

B.4.7.1 Generate Terminal Pathname

L_ctermid must be defined appropriately for a given implementation and must be greater than zero so that array declarations using it are accepted by the compiler. The value includes the terminating null byte.

B.4.7.2 Determine Terminal Device Name

The term "terminal" is used instead of the historical term "terminal device" in order to avoid a reference to an undefined term.

B.4.8 Configurable System Variables

This section was added in response to requirements of application developers, and particularly the X/OPEN system vendors. It is closely related to **Configurable Pathname Variables** §B.5.7 as well.

Although a portable application can run on all systems by never demanding more resources than the minimum values published in the standard, it is useful for that application to be able to use the actual value for the quantity of a resource available on any given system. To do this, the application will make use of the value of a symbolic constant in **<limits.h>** or **<unistd.h>**.

However, once compiled, the application must still be able to cope if the amount of resource available is increased. To that end, an application may need a means of determining the quantity of a resource, or the presence of an option, at execution time.

Two examples are offered:

> Applications may wish to act differently on systems with or without the Job Control Option. Applications vendors who wish to distribute only a single binary package to all instances of a computer architecture would be forced to assume job control is never available if it were to rely solely on the **<unistd.h>** value published in the standard.

> International applications vendors occasionally require knowledge of the {CLK_TCK} value. Without the facilities of this section, they would be required to either distribute their applications partially in source form or to have 50 Hertz and 60 Hertz versions for the various countries they do business in.

It is the understanding that many applications are actually distributed widely in executable form that lead to this facility. If limited to the most restrictive values in the headers, such applications would have to be prepared to accept the most limited environments offered by the smallest microcomputers. Although this is entirely portable, it was felt by the Working Group that they should be able to take advantage of the facilities offered by large systems, without the restrictions associated with source and object distributions.

During the very heated arguments that accompanied the discussions of this feature, it was pointed out that it is almost always possible for an application to discern what a value might be at runtime by suitably testing the waters. And, in any event, it could always be written to adequately deal with error returns from the various functions. In the end, it was felt that this imposed an unreasonable level of complication and sophistication on the application writer.

This runtime facility is not meant to provide ever-changing values that applications will have to check multiple times. The values are seen as changing no more frequently than once per system initialization, such as by a system administrator or operator with an automatic configuration program. The standard specifies that they shall not change within the lifetime of the process.

Some values apply to the system overall and others vary at the file system or directory level. These latter are described in **Configurable Pathname Variables** §B.5.7.

B.4.8.1 Get Configurable System Variables

Note that all values returned must be expressable as integers. The Working Group considered using string values, but the additional flexibility of this approach was rejected due to its added complexity of implementation and use.

Some values, such as {PATH_MAX}, are sometimes so large that they must not be used to, say, allocate arrays. The *sysconf*() function will return a negative value to show that this symbolic isn't even defined, in this case.

B.5 Files and Directories

See **pathname resolution** §2.4.

B.5.1 Directories

Historical implementations prior to 4.2BSD had no special functions, types, or headers for directory access. Instead, directories were read with *read*() §6.4.1 and each program that did so had code to understand the internal format of directory files. Many such programs did not correctly handle the case of a maximum-length (historically fourteen character) filename and would neglect to add a null character string terminator when doing comparisons. The access methods in the standard eliminate that bug, as well as hiding differences in implementations of directories or file systems.

The directory access functions as described in an Appendix of the POSIX Trial Use Standard were derived from 4.2BSD, were adopted in System V Release 3 and are in *SVID* Volume 3, with the exception of a type difference for the *d_ino* field. That field represents implementation-dependent or even file system-dependent information (the i-node number in most implementations). Since the directory access mechanism is intended to be implementation independent, and since only system programs, not ordinary applications, need to know about the i-node number (or **file serial number** §2.3) in this context, the *d_ino* field does not appear in the present standard. Also, programs that want this information can get it with *stat*() §5.6.2.

B.5.1.1 Format of Directory Entries

Information similar to that in the header **<dirent.h>** is contained in a file **<sys/dir.h>** in 4.2BSD and 4.3BSD. The equivalent in these implementations of *struct dirent* from the standard is *struct direct*. The filename was changed because the name **<sys/dir.h>** was also used in earlier implementations to refer to definitions related to the older access method; this produced name conflicts. The name of the structure was changed because the standard does not completely define what is in the structure, so it could be different on some implementations from *struct direct*.

The name of a character array of an unspecified size should not be used as an *lvalue*. Use of

```
sizeof (d_name)
```

is incorrect; use

```
strlen (d_name)
```

instead.

This description of the *d_name* element was changed because the previous version gave the impression that the character array *d_name* was of a fixed size. Implementations may need to declare *struct dirent* with an array size for *d_name* of 1, but the actual number of characters provided matches (or only slightly exceeds) the length of the file name.

Currently, implementations are excluded if they have *d_name* with type char *. Lacking experience of such implementations, the Working Group declined to try to describe in standards language what to do if either type were permitted.

B.5.1.2 Directory Operations

The returned value of *readdir*() merely *represents* a directory entry. No equivalence should be inferred.

Since *readdir*() returns NULL both

1. when it detects an error and
2. when the end of the directory is encountered,

an application that needs to tell the difference must set *errno* to zero before the call and check it if NULL is returned. Because the function must not change *errno* in case 2 and must set it to a non-zero value in case 1, zero *errno* after a call returning NULL indicates end of directory, otherwise an error:

Routines to deal with this problem more directly were proposed.

int derror (*dirp*)
DIR **dirp*;

void clearderr (*dirp*)
DIR **dirp*;

The first would indicate whether an error had occurred, and the second would clear the error indication. The simpler method involving *errno* was adopted instead by requiring that *readdir*() not change *errno* when end of directory is encountered.

Historical implementations include two more functions.

```
long telldir (dirp)
DIR *dirp;

void seekdir (dirp, loc)
DIR *dirp;
long loc;
```

The *telldir*() function returns the current location associated with the named directory stream.

The *seekdir*() function sets the position of the next *readdir*() operation on the directory stream. The new position reverts to the one associated with the directory stream when the *telldir*() operation was performed.

These functions have restrictions on their use related to implementation details. Their capability can also be accomplished by saving a filename found by *readdir*() and later using *rewinddir*() and a loop on *readdir*() to relocate the position from which the filename was saved. Though this method is probably slower than using *seekdir*() and *telldir*(), there are few applications in which the capability is needed. For these reasons, the Working Group decided not to include *seekdir*() and *telldir*() in the standard.

An error or signal indicating that a directory has changed while open was considered but rejected.

B.5.2 Working Directory

B.5.2.1 Change Current Working Directory

B.5.2.2 Working Directory Pathname

Since the maximum pathname length is arbitrary unless {PATH_MAX} is defined, an application cannot supply a *buf* with *size* {{PATH_MAX} + 1} in general.

Having the routine take no arguments and instead use the C function *malloc*() to produce space for the returned argument was considered. The advantage is that *getcwd*() knows how big the working directory pathname is and can allocate an appropriate amount of space. But the programmer would have to use the C function *free*() to free the resulting object, or each use of *getcwd*() would further reduce the available address space. Also, *malloc*() and *free*() are used nowhere else in the present standard. Finally, *getcwd*() is taken from the *SVID*, where it has the two arguments used in the standard.

The older function *getwd*() was rejected for use in this context because it had only a buffer argument and no size argument, and thus had no way to prevent overwriting the buffer, except to depend on the programmer to provide a large enough buffer.

The result if a NULL argument is passed to *getcwd*() is left implementation defined because some implementations dynamically allocate space in that case.

If a program is operating in a directory where some (grand)parent directory does not permit reading, *getcwd*() may fail, as in most implementations it must read the directory to determine the name of the file. This can occur if search but not read permission is granted in an intermediate directory, or if the program is placed in that directory by some more priveleged process (e.g. `login`). Including this error makes the reporting of the error consistent, and warns the application writer that *getcwd*() can fail for reasons beyond his control. (The other two failures should not be beyond his control.) Some implementations can avoid this occurrence (e.g. by implementing *getcwd*() using *pwd*(), and making *pwd*() a set-user-root process), thus the error was made optional.

Because the standard permits the addition of other errors, this would be a common addition and yet one that applications could not be expected to deal with without this addition.

B.5.3 General File Creation

B.5.3.1 Open a File

Except as specified in the standard, the flags allowed in *oflag* are not mutually exclusive and any number of them may be used simultaneously.

See getgroups §B.4.2.3 about the group of a newly-created file.

The use of *open*() §5.3.1 to create a regular file is preferable to the use of *creat*() §5.3.2 because the latter is redundant and included only for historical reasons.

Implementations may deny access and return [EACCES] for reasons other than just those listed in the [EACCES] definition.

B.5.3.2 Create a New File or Rewrite an Existing One

This interface is redundant. Its services are also provided by the *open*() function. It has been included primarily for historical purposes since many existing applications depend on it.

B.5.3.3 Set File Creation Mask

Unsigned argument and return types for *umask*() were proposed. The return type was left unchanged, but the argument was changed to *mode_t* §B.2.6.

B.5.3.4 Link to a File

See directory entry §B.2.3.

Linking to a directory is restricted to the super-user in most historical implementations because this capability may produce loops in the file hierarchy or otherwise corrupt the file system. However, file system implementations may be envisioned where multiple parents of a directory are handled without adverse side effects. Therefore, the standard does not require the restriction to the super-user. But see *rename*() §B.5.5.3. See also *unlink*() §5.5.1.

B.5.4 Special File Creation

B.5.4.1 Make a Directory

See *mode_t* §B.2.6.

This function originated in 4.2BSD and was added to System V in Release 3.0, following the Trial Use Standard.

4.3BSD detects [ENAMETOOLONG].

See **getgroups** §B.4.2.3 about the group of a newly-created directory.

B.5.4.2 Make a FIFO Special File

The syntax of this routine is intended to maintain compatibility with existing implementations of *mknod*(). The latter function was included in the *1984 /usr/group Standard*, but only for use in creating FIFO special files. The *mknod*() function was excluded from POSIX as implementation defined and replaced by *mkdir*() §5.4.1 and *mkfifo*() §5.4.2.

See **getgroups** §B.4.2.3 about the group of a newly-created FIFO.

B.5.5 File Removal

Although *rmdir*() and *rename*() originated in 4.2BSD, the behavior specified for when the directory to be removed does not exist or *new* already exists (returning [EEXIST] in *errno*) is not compatible with 4.2BSD or 4.3BSD, which return [ENOTEMPTY]. Therefore, either value is allowed by the standard. The function was added to System V in Release 3.0 but uses [ENOENT] where the standard uses [ENAMETOOLONG]. Volume 3 of the *SVID*, page 129, states: "FUTURE DIRECTION: To conform with the IEEE POSIX standard, when it is adopted as a full-use standard, the value of *errno* indicating that ..."

The Berkeley implementations of *rmdir*() and *rename*() used [ENOTEMPTY] for this error condition. When the /usr/group Standard was published, it contained [EEXIST] instead. When AT&T adopted these functions into System V, they used the /usr/group Standard as their reference. Therefore, several existing applications and implementations support/use both forms and the Working Group could not agree on either value. All implementations are required to supply both [EEXIST] and [ENOTEMPTY] in **<errno.h>** with distinct values so that applications can use both values in C language case statements.

B.5.5.1 Remove Directory Entries

Unlinking a directory is restricted to the super-user in many historical implementations for reasons given in *link*() §B.5.3.4. But see *rename*() §B.5.5.3.

B.5.5.2 Remove a Directory

See also [ENOTEMPTY] and [ENAMETOOLONG] §B.5.5.

B.5.5.3 Rename a File

This *rename*() call is equivalent for regular files to that defined by the C Standard. Its inclusion here expands that definition to include actions on directories and specifies behavior when the *new* parameter names a file that already exists. That specification requires that the action of the function be atomic.

One of the reasons for introducing this function was to have a means of renaming directories while permitting implementations to prohibit the use of *link*() §5.3.4 and *unlink*() §5.5.1 with directories, thus constraining links to directories to those made by *mkdir*() §5.4.1.

The specification that if *old* and *new* refer to the same file describes existing, although undocumented, 4.3BSD behavior. It is intended to guarantee that:

```
rename("x", "x");
```

does not remove the file.

Renaming dot or dot-dot is prohibited in order to prevent cyclical file system paths.

See also [[ENOTEMPTY] and [ENAMETOOLONG] §B.5.5.

B.5.6 File Characteristics

The function *ustat*(), which appeared in the *1984 /usr/group Standard* and is still in the *SVID*, was removed from the present standard before Trial Use because it was:

- Not reliable. The amount of space available can change between the time the call is made and the time the calling process attempts to use it.
- Not required. The only known program that uses it is the text editor *ed*.

It was also not readily extensible to networked systems.

B.5.6.1 File Characteristics: Header File and Data Structure

See *dev_t* §B.2.6, *link_t* §B.2.6, *mode_t* §B.2.6, *off_t* §B.2.6, and *uid_t* §B.2.6.

The S_ISUID and S_ISGID bits may be cleared on any write, not just on *open*() §5.3.1, as some historical implementations do it.

System calls that update the time entry fields in the *stat* structure must be documented by the implementors. It is not expected that routines that call one of these system calls need to document this as a side effect. (Note that this includes most of the *stdio* routines in the *ANSI/X3.159-198x Programming Language C Standard.*) POSIX conforming systems should not update the time entry fields for functions listed in the standard unless the standard requires that they do, except in the case of documented extensions to the standard.

B.5.6.2 Get File Status

The intent of the paragraph describing "additional implementation defined access constraints" is to allow a secure implementation where a process with a label that does not dominate the file's label cannot perform a *stat*() function. This is not related to read permission; a process with a label that dominates the file's label will not need read permission. An implementation that supports write-up operations could fail *fstat*() function calls even though it has a valid file descriptor open for writing.

B.5.6.3 File Accessibility

Some Working Group discussions centered around inadequacies in the *access*() function led to the creation of an *eaccess*() function because:

1. Historical implementations of *access*() don't test file access correctly when the process's real user ID is super-user. In particular, they always return zero when testing execute permissions without regard to whether the file is executable.

2. The super-user has complete access to all files on a system. As a consequence, programs started by the super-user and switched to the effective user ID with lesser privileges cannot use *access*() to test their file access permissions.

After *eaccess*() was reviewed, the Working Group found that it still didn't resolve problem 1, so the standard now allows *access*() to behave in the desired way because several implementations have corrected the problem. It was also argued that problem 2 is more easily solved by using *open*(), *chdir*(), or *exec*() functions as appropriate and responding to the error there, rather than creating a new function that wouldn't be as reliable. Therefore, *eaccess*() was taken back out of the standard.

Secure implementations will probably need an extended *access*()-like function, but the Working Group did not have enough of the requirements to define it yet. This could be proposed as an extension to the Full Use standard.

The phrase "an implementation may subsitute search permissions for execute permission" reflects the two possibilities implemented by historical implementations when checking super-user access for X_OK.

B.5.6.4 Change File Modes

B.5.6.5 Change Owner and Group of File

System III and System V allow a user to give away files, that is, the owner of a file may change its user ID to anything. This is a serious problem for implementations which are intended to meet government security regulations. Version 7 and 4.3BSD permit only the super-user to change the user ID of a file. Some government agencies (usually not ones concerned directly with security) find this limitation too confining. The standard uses "may" to permit secure implementations while not disallowing System V.

System III and System V allow the owner of a file to change the group ID to anything. Version 7 permits only the super-user to change the group ID of a file. 4.3BSD permits the owner to change the group ID of a file to its effective group ID or to any of the groups in the list of supplementary group IDs, but to no others.

The decision to require that, for non-privileged processes, the S_ISUID and S_ISGID bits be cleared on regular files but only *may* be cleared on non-regular files was to allow plans for using these bits in implementation specified manners on directories. Similar cases could be made for other file types, so the standard does not require that these bits be cleared except on regular files. Note that as these cases arise, the system implementors will have to determine whether these features enable any security loopholes and specify appropriate restrictions.

B.5.6.6 Set File Access and Modification Times

The *actime* structure member must be present, so that an application may set it, even though an interface implementation may ignore it and not change the access time on the file. If an application intends to leave one of the times of a file unchanged while changing the other, it should use *stat*() §5.6.2 to retrieve the file's st_atime §5.6.1.2.2 and st_mtime §5.6.1.2.2 parameters, set *actime* and *modtime* in the buffer, and change one of them before making the *utime*() call.

B.5.7 Configurable Pathname Variables

When the runtime facility described in Configurable Pathname Variables §B.4.8 was designed, it was realized that some variables change depending on the file system. For example, it is quite feasible for a system to have two varieties of file systems mounted: a System V, and; a Berkeley "Fast File System."

If limited to strictly compile-time features, no application that was widely distributed in executable binary form could rely on more than 14 bytes in a pathname component, as that is the minimum published for {NAME_MAX} in this standard. The *pathconf*() function allows the application to take advantage of the most liberal file system available at runtime. In many Berkeley-based systems, 255 bytes are allowed for pathname components.

These values are potentially changeable at the directory level, not just at the file system. And, unlike the overall system variables, there is no guarantee that these might not change during program execution. However, if the program is dealing with an open file descriptor, using the *fpathconf*() function, they won't change while the file is still open.

B.5.7.1 Get Configurable Pathname Variables

The *pathconf*() function was proposed immediately after the *sysconf*() function when it was realized that some configurable values may differ across file system, directory, or device boundaries.

For example, {NAME_MAX} frequently changes between System V and BSD-based file systems; System V uses a maximum of 14, Berkeley 255. On an implementation that provided both types of file systems, an application would be forced to limit all pathname components to 14 bytes, as this would be the value specified in **<limits.h>** on such a system.

Therefore, various useful values can be queried on any pathname or file descriptor, assuming that the appropriate permissions are in place.

Note that, unlike the values returned by *sysconf*(), the pathname-oriented variables are potentially more volatile and are not guaranteed to remain constant throughout the process's lifetime. For example, in between two calls to *pathconf*() the file system in question may have been unmounted and remounted with different characteristics.

B.6 Input and Output Primitives

Rationale for the Change from O_NDELAY to O_NONBLOCK.

System III and System V have included a flag, O_NDELAY, to mark file descriptors so that user processes would not block when doing I/O to them. If the flag is set, a *read*() §6.4.1 or *write*() §6.4.2 call which would otherwise need to block for data returns a value of zero instead. But a *read*() call also returns a value of zero on end of file, and applications have no way to distinguish between these two conditions.

BSD systems support a similar feature through a flag with the same name, but somewhat different semantics. The flag applies to all users of a file (or socket) rather than only to those sharing a file descriptor. The BSD interface provides a solution to the problem of distinguishing between a blocking condition and an end of file condition by returning an error, [EWOULDBLOCK], on a blocking condition.

The *1984 /usr/group Standard* includes an interface with some features from both AT&T and BSD. The overall semantics are that it applies only to a file descriptor. However, the return indication for a blocking condition is an error, [EAGAIN]. This was the starting point for POSIX.

The problem with the *1984 /usr/group Standard* that it does not allow compatibility with existing applications. An implementation cannot both conform to this standard and support applications written for existing AT&T or BSD systems. This was the cause of at least one objection during the trial-use balloting. Several changes have been considered, either at that time or more recently, to address this issue. These include:

0) no change (from *1984 /usr/group Standard*)

1) changing to System III/V semantics

2) changing to BSD semantics

3) broadening the standard to allow conforming implementation a choice among these semantics

4) changing the name of the flag from O_NDELAY

5) changing to System III/V semantics and providing a new call to distinguish between blocking and end of file conditions

The consensus of the Working Group at the January, 1986, meeting in Denver, was that c (4) is the best alternative. The new name is O_NONBLOCK. This alternative allows a conforming implementation to provide backward compatibility at the source and/or object level with either AT&T or BSD systems (but the standard does not require or even suggest that this be done). It also allows Conforming Application Using Extensions the functionality to distinguish between blocking and end of file conditions, and to do so in as simple a manner as any of the alternatives. The greatest shortcoming was that it forces all existing AT&T and BSD applications that use this facility to be modified in order to c strictly conform to the standard. This same shortcoming applies to (0) and (3) as well, and it applies to one group of applications for (1), (2), and (5).

Systems may choose to implement both O_NDELAY and O_NONBLOCK, and there is no conflict as long as an application does not turn both flags on at the same time.

See also scope §B.6.5.1.

B.6.1 Pipes

The requirement that attempts to write on *fildes*[0] or to read on *fildes*[1] shall fail does not make the 4.3BSD implementation of pipes as sockets nonconforming, since the pipe code carefully sets up a pair of unidirectional sockets. System V Release 3 as distributed does not use streams for pipes. The historical (Version 7) error for such an attempt is [EBADF]

B.6.1.1 Create an Inter-Process Channel

The wording carefully avoids using the verb "to open" in order to avoid any implication of use of *open*() §5.3.1.

See also Write to a Pipe §B.6.4.2. A

B.6.2 File Descriptor Manipulation

B.6.2.1 Duplicate an Open File Descriptor

These interfaces are redundant. Their services are also provided by the *fcntl*() function. 9
They have been included in this standard primarily for historical reasons, since many 9
existing applications use them. 9

In the description of [EBADF] the case of *fildes* being out of range is covered by the given case of *fildes* not being valid. The descriptions for *fildes* and *fildes2* are different because the only kind of invalidity that is relevant for *fildes2* is whether it is out of range, that is, it does not matter whether *fildes2* refers to an open file when the *dup2*() call is made.

If *fildes2* is a valid file descriptor, it shall be closed, regardless of whether the function c
returns an indication of success or failure, unless *fildes2* is equal to *fildes*. c

B.6.3 File Descriptor Deassignment

B.6.3.1 Close a File

Once a file is closed, the file descriptor no longer exists, since the integer corresponding to it no longer refers to a file.

B.6.4 Input and Output

The standard permits return of the number of bytes read or written after an interrupted operation in order to promote compatibility with System V, even though it makes writing a Conforming Application more difficult.

Whether the return values of, and *nbyte* arguments to, *read*() §6.4.1 and *write*() §6.4.2 should be signed or unsigned was a chronic source of controversy. On machines where type **int** is of sixteen bits, only 32767 bytes may be transferred on one function call. If *nbyte* were unsigned, it would be convenient for the return value to be of the same type. But if the returned value were unsigned, it would be necessary to compare it to (unsigned)−1 in order to detect an error. Although a definition such as IO_ERR could be provided to simplify code, still many existing applications would not conform.

The Working Group decided to make *nbyte* unsigned, with the results of use of values greater than {INT_MAX} (often 32767) being made implementation defined. However, the return value was left signed to avoid the error-detection problem. It is still possible to compare the return value directly with *nbyte*, since the C Standard specifies that the comparison will be done unsigned.

Use of the type **long** was considered in order to avoid the sixteen bit problem, but not adopted.

New functions like *read*() and *write*() called *lread*() and *lwrite*() and differing only in that their *nbyte* argument and return values would be of type *off_t* §2.8 were proposed but rejected. The Working Group is not necessarily against the creation of *lread*() and *lwrite*() calls, but was unable to clearly identify the need given the above. It was also noted that C has similar constraints parallel to those mentioned above, and that the type of *sizeof* is not necessarily **long** (where the largest object cannot exceed `sizeof(char[MAXINT])`.

There were recommendations to add format parameters to *read*() and *write*() in order to handle networked transfers among heterogenous file system and base hardware types. Such a facility may be required for support by the OSI presentation of layer services. However, the Working Group determined that this should correspond with similar C Language facilities, and that is beyond the scope of the 1003 effort. The concept was suggested to X3J11 for their consideration as a possible area for future work.

In 4.3BSD, a signal does not interrupt a *read*() §6.4.1 or a *write*() §6.4.2; thus the notes below regarding *setjmp*() §8.3.1 and *longjmp*() §8.3.1. In 4.2BSD, 4.3BSD, and Version 8 there is an additional function, *select*() §B.6.4, whose purpose is to pause until specified activity (data to read, space to write, etc.) is detected on specified file descriptors. It is common in applications written for those systems for *select*() to be used before *read*() in situations (such as keyboard input) where interruption of I/O due to a signal is desired. But this approach does not conform, because *select*() is not in the standard. The Working Group included *setjmp*() and *longjmp*() so that there would be a method usable by Conforming Application Using Extensions. 4.3BSD semantics are permitted by not requiring the implementation to return [EINTR] on a *read*() or *write*().

The standard permits *read*() and *write*() to return the number of bytes successfully transferred when interrupted by an error. This is not required because it is incompatible with Version 7, System III, and System V.

B.6.4.1 Read from a File

The file offset is not incremented if an error is returned.

References to actions taken on an "unrecoverable error" have been removed. It is considered beyond the scope of this standard to describe what happens in the case of hardware errors.

B.6.4.2 Write to a File

An attempt to write to a pipe or FIFO has several major characteristics:

Atomic/non-atomic

> A write is atomic if the whole amount written in one operation is not interleaved with data from any other process. This is useful when there are multiple writers sending data to a single reader. Applications need to know how large a write request can be expected to be performed atomically. We call this maximum {PIPE_BUF}. The standard does not say whether write requests for more than {PIPE_BUF} bytes will be atomic, but requires that writes of {PIPE_BUF} or less bytes shall be atomic.

Blocking/immediate

Blocking is only possible with O_NONBLOCK clear. If there is enough space for all the data requested to be written immediately, the implementation should do so. Otherwise, the process may block, that is, pause until enough space is available for writing. The effective size of a pipe or FIFO (the maximum amount that can be written in one operation without blocking) may vary dynamically, depending on the implementation, so it is not possible to specify a fixed value for it.

Complete/partial/deferred

A write request,

```
int fildes, nbyte, ret;
char *buf;

ret = write(fildes, buf, nbyte);
```

may return

complete: ret = *nbyte*

partial: ret < *nbyte*
This shall never happen if *nbyte* ≤ {PIPE_BUF}. If it does happen (with *nbyte* > {PIPE_BUF}), the standard does not guarantee atomicity, even if *ret* ≤ {PIPE_BUF}, because atomicity is guaranteed according to the amount *requested*, not the amount written.

deferred: ret = −1, errno = [EAGAIN]
This error indicates that a later request may succeed. It does not indicate that it *shall* succeed, even if *nbyte* ≤ {PIPE_BUF}, because if no process reads from the pipe or FIFO, the write will never succeed. An application could usefully count the number of times [EAGAIN] is caused by a particular value of *nbyte* > {PIPE_BUF} and perhaps do later writes with a smaller value, on the assumption that the effective size of the pipe may have decreased.

Partial and deferred writes are only possible with O_NONBLOCK set.

The relations of these properties are best shown in tables.

Write to a Pipe or FIFO with O_NONBLOCK clear.			
immediately writable:	none	some	*nbyte*
nbyte ≤ {PIPE_BUF}	atomic blocking *nbyte*	atomic blocking *nbyte*	atomic immediate *nbyte*
nbyte > {PIPE_BUF}	blocking *nbyte*	blocking *nbyte*	blocking *nbyte*

If the O_NONBLOCK flag is clear, a write request shall block if the amount writable immediately is less than that requested. If the flag is set (by *fcntl*()), a write request shall never block.

Write to a Pipe or FIFO with O_NONBLOCK set.			
immediately writable:	none	some	*nbyte*
nbyte ≤ {PIPE_BUF}	−1, [EAGAIN]	−1, [EAGAIN]	atomic *nbyte*
nbyte > {PIPE_BUF}	−1, [EAGAIN]	< *nbyte* or −1, [EAGAIN]	≤ *nbyte* or −1, [EAGAIN]

There is no way provided for an application to determine whether the implementation will ever perform partial writes to a pipe or FIFO. Every application should be prepared to handle partial writes when O_NONBLOCK is set and the requested amount is greater than {PIPE_BUF}, just as every application should be prepared to handle partial writes on other kinds of file descriptors.

Where the standard requires −1 returned and **errno** set to [EAGAIN], most historical implementations return 0 (with the O_NDELAY flag set: that flag is the historical predecessor of O_NONBLOCK, but is not itself in the standard). The error indications in the standard were chosen so that an application can distinguish these cases from end of file. While *write*() cannot receive an indication of end of file, *read*() can, and the Working Group chose to make the two functions have similar return values. Also, some existing systems (e.g., Version 8) permit a write of zero bytes to mean that the reader should get an end of file indication: for those systems, a return value of zero from write indicates a successful write of an end of file indication.

The concept of a {PIPE_MAX} limit (indicating the maximum number of bytes that can be written to a pipe in a single operation) was discussed by the Working Group. The Group decided this concept would unnecessarily limit application writing.

See also O_NONBLOCK §B.6.

The file offset is not incremented if an error is returned.

The standard does not specify behavior of concurrent writes to a file from multiple processes. Applications should use some form of concurrency control.

References to actions taken on an "unrecoverable error" have been removed. It is considered beyond the scope of this standard to describe what happens in the case of hardware errors.

B.6.5 Control Operations on Files

B.6.5.1 Data Definitions for File Control Operations

The main distinction between the file descriptor flags and the file status flags is scope. The former apply to a single file descriptor only, while the latter apply to all file descriptors that share a common open file description (by inheritance through *fork*() §3.1.1 or an F_FDUPFD operation with *fcntl*() §6.5.2). Neither apply to file descriptors that have different file pointers even if they refer to the same file (by separate *open*() §5.3.1 calls). For O_NONBLOCK, this scoping is like that of O_NDELAY in System V rather than in 4.3BSD, where the scoping for O_NDELAY is different from all the other flags accessed via the same commands.

For example:

```
fd1 = open (pathname, oflags);
fd2 = dup (fd1);
fd3 = open (pathname, oflags);
```

Does an *fcntl*() call on *fd1* also apply to *fd2* or *fd3* or to both? According to the standard, F_SETFD applies only to *fd1*, while F_SETFL applies to *fd1* and *fd2* but not to *fd3*. This is in agreement with all common historical implementations except for BSD with the F_SETFL command and the O_NDELAY flag (which would apply to *fd3* as well). Note that this does not force any incompatibilities in BSD implementations, because O_NDELAY is not in the standard. See also O_NONBLOCK §B.6.

B.6.5.2 File Control

The ellipsis in the Synopsis is the syntax specified by the C Standard for a variable number of arguments. It is used because System V uses pointers for the implementation of file locking functions.

POSIX permits concurrent read and write access to file data using the *fcntl*() function; this is a change from the /usr/group Standard and previous drafts, which included a *lockf*() function. Without concurrency controls, this feature may not be fully utilized without occasional loss of data. Since other mechanisms for creating critical regions, such as semaphores, are not included, a file record locking mechanism was thought appropriate. The *fcntl*() mechanism may be used to implement semaphores, although access is not first-in-first-out without extra application implementation effort.

Data losses occur in several ways. One is that read and write operations are not atomic, and as such a reader may get segments of new and old data if concurrently written by another process. Another occurs when several processes try to update the same record, without sequencing controls; several updates may occur in parallel and the last writer will "win." Another case is a b-tree or other internal list-based database that is undergoing reorganization. Without exclusive use to the tree segment by the updating process, other reading processes chance getting lost in the database when the index blocks are split, condensed, inserted, or deleted. While *fcntl*() is useful for many applications, it is not intended to be overly general, and will not handle the b-tree example well.

This facility is only required for regular files, because it is not appropriate for many devices such as terminals and network connections. However, if it is not supported on a given device, the *fcntl*() function must return an error of [ENODEV]

Since *fcntl*() works with "any file descriptor associated with that file, however it is obtained," the file descriptor may have been inherited through a *fork*() §3.1.1 or *exec* §3.1.2 operation and thus may affect a file that another process also has open.

The use of the open file description to identify what to lock requires extra calls and presents problems if several processes are sharing a open file description but there are too many implementations of the existing mechanism for the standard to use different specifications.

But note that while a open file description may be shared through *fork*(), locks are not inherited through *fork*(). Yet locks may be inherited through *exec*().

Shared read locks are not part of the design because no easy implementation was seen that would eliminate the race conditions and lockout that would occur in normal usage.

Since locking is performed with *fcntl*(), rather than *lockf*(), this specification prohibits use of locking on a file that is not open for writing.

Before successful return from a F_SETLK or F_SETLKW request, the previous lock type for each byte in the specified region shall be replaced by the new lock type. This can result in a previously locked region being split into smaller regions. If this would cause the number of regions being held by all processes in the system to exceed a system-imposed limit, the *fcntl*() function returns −1 with *errno* set to [ENOLCK].

Mandatory locking was a major feature of the *1984 /usr/group Standard*. For advisory file record locking to be effective, all processes that have access to a file must cooperate and use the advisory mechanism before doing I/O on the file. Enforcement-mode record locking is important when it cannot be assumed that all processes are cooperating. For example, if one user uses an editor to update a file at the same time that a second user executes another process that updates the same file, if only one of the two processes is using advisory locking, the processes are not cooperating. Enforcement mode record locking would protect against accidental collisions.

Secondly, advisory record locking requires a process using locking to bracket each I/O operation with lock (or test) and unlock operations. With enforcement mode file and record locking, a process can lock the file once and unlock when all I/O operations have been completed. Eforcement mode record locking provides a base that can be enhanced, for example, with shareable locks. That is, the mechanism could be enhanced to allow a process to lock a file so other processes could read it but none of them could write it.

Mandatory locks were omitted for several reasons.

1. Mandatory lock setting was done by multiplexing the setgid bit in most implementations; this was confusing, at best.

2. Relationship to file truncation as supported in 4.2BSD was not well specified.

3. Any publicly readable file could be locked by anyone. Many historical implementations keep the password database in a publicly-readable file. A malicious user could thus prohibit logins. Another possibility would be to hold open a long-distance telephone line.

4. Some demand-paged historical implementations offer memory mapped files, and enforcement cannot be done on that type of file.

Since sleeping on a region is interrupted with any signal, *alarm*() §3.4.1 may be used to provide a timeout facility in applications requiring it. This is useful in deadlock detection. Although the *fcntl*() implementation must provide deadlock detection between processes that are related by locked resources, it does not have to account for deadlocks caused by activities unrelated to *fcntl*() that have suspended a lock owner.

The *l_start* element of the *flock* structure and the *offset* argument of *lseek*() are, in some cases, taken as signed offsets from some position in a file, but the type of these objects is allowed to be unsigned. This apparent conflict is avoided by the C Standard's definitions of conversions from signed to unsigned and of arithmetic operations on unsigned types. If U is of type *off_t*, the expressions

```
U + ((off_t) (-i))
```

and

```
U - i
```

will produce the same result, and, for example,

```
lseek (fd, (off_t) - 4, SEEK_END);
```

is well defined.

B.6.5.3 Reposition Read/Write File Offset

The C Standard includes the functions *fgetpos*() §B.6.5.3 and *fsetpos*() §B.6.5.3 which work on very large files by use of a special positioning type.

Although *lseek*() may position the file offset beyond the end of the file, this function does not itself extend the size of the file. While the only function in POSIX that may extend the size of the file is *write*() §6.4.2, several Standard C functions, such as *fwrite*(), *fprintf*(), etc., may do so (by causing calls on *write*()).

An illegal file offset that would cause [EINVAL] to be returned may be both implementation defined and device dependent (for example, memory may have few illegal values). A negative file offset may be legal for some devices in some implementations.

See *fcntl*() §B.6.5.2 for a explanation of the use of signed and unsigned offsets with *lseek*().

B.7 Device- and Class-Specific Functions

This section has probably undergone more debate and revision than any other in the standard. Numerous historical implementations were investigated, and at least four major proposals were made.

There are several sources of the difficulties of this section:

- The basic Version 7 *ioctl*() mechanism is difficult to specify adequately, due to its use of a third argument that varies in both size and type according to the second, command, argument.
- System III introduced and System V continued *ioctl*() commands that are completely different from those of Version 7.
- 4.2BSD and other Berkeley systems added to the basic Version 7 *ioctl*() command set; some of these were for features such as job control that POSIX eventually adopted.
- None of the basic historical implementations are adequate in an international environment. This concern is not technically within the scope of POSIX, but the Working Group did not want to supply unnecessary impediments to internationalization.

The *1984 /usr/group Standard* attempted to specify a portable mechanism that application writers could use to get and set the modes of an asynchronous terminal. The intention of that committee was to provide an interface that was neither implementation specific nor hardware dependent. Initial proposals dealt with high level routines similar to the *curses* library (available on most historical implementations). In such an implementation, the user interface would consist of calls similar to:

setraw();
setcooked();

It was quickly pointed out that if such routines were standardized, the definition of "raw" and "cooked" would have to be provided. If these modes were not well defined in the standard, application code could not be written in a portable way. However, the definition of the terms would force low level concepts to be included in a supposedly high level interface definition.

Recognizing the pitfalls of the high level approach, the Working Group focused on the necessary low level attributes that were needed to support the necessary terminal characteristics (e.g., line speeds, raw mode, cooked mode, etc.). After considerable debate, a structure similar to, but more flexible than, the AT&T System III *termio* was agreed upon. The format of that structure, referred to as the *termios* structure, has formed the basis for the current section.

A method is needed to communicate with the system about the *termios* information. Proposals have included:

The *ioctl*() function

as in System V. This has the same problems as mentioned above for the Version 7 *ioctl*() function, and is basically identical to it. Another problem is that the direction of the command (whether information is written from or read into the third argument) is not specified: in historical implementations only the device driver knows for sure. This is a problem for networked implementations. It is also a problem that there is no size parameter to specify the variable size of the third argument, and similarly for its type.

An *iocntl*() function

with additional arguments specifying direction, type, and size. But these new arguments would not help application writers, who would have no control over their values, which would have to match each command exactly. The new arguments do, however, solve the problems of networked implementations. And *iocntl*() is implementable in terms of *ioctl*() on historical implementations (without need for modifying existing code), although it is easy to update existing code to use the arguments directly.

A *termcntl*() function

with the same arguments as proposed for the *iocntl*() function. The difference would be that *termcntl*() would be limited to terminal interface functions: there would be other interface functions, such as a *tapecntl*() function for tape interfaces, rather than a single general device interface routine.

Unspecified functions

The issue of what the interface function(s) should be called was sidestepped for some time after the Trial Use Standard while the Working Group concentrated on the details of the information to be handled. The resulting specification resembles the information in System V, but attempts to avoid problems of case, speed, networks, and internationalization.

Specific *tc_**() functions

to replace each *ioctl*() function were finally incorporated into the standard, instead of any of the above-mentioned proposals.

The issue of modem control [Unknown Reference Type] § was excluded from POSIX on the grounds that:

1. it was concerned with setting and control of hardware timers, and

2. the appropriate timers and settings vary widely internationally.

3. Feedback from X/OPEN indicated that this facility was not consistent with Europeon needs, and that specification of such a facility was not a requirement for portability from their "international perspective."

B.7.1 General Terminal Interface

Although the Working Group attempted to take into account needs of both interface implementors and application developers throughout the standard, more attention was paid to the needs of the latter in this section. 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, etc. These functions should usually be done by the operating system, possibly at interrupt level.

The *tc**() functions were introduced as a way of avoiding the problems inherent in the traditional *ioctl*() §B.7.1 function and in variants of it that were proposed. For example, *tcgets*() is specified in place of the use of the TCGETS *ioctl*() command function. This allows specification of all the arguments in a manner consistent with the 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.

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*() §6.5.2 is broken.

B.7.1.1 Interface Characteristics

B.7.1.1.1 Description

B.7.1.1.2 Opening a Terminal Device File

B.7.1.1.3 Process Groups

B.7.1.1.4 The Controlling Terminal

B.7.1.1.5 Job Access Control

The foreground/background check performed by the terminal driver must be repeatedly performed until the calling process moves into the foreground. 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 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 checks whenever a process is about to access the terminal. In the case of *write*() or the **Control Functions** §7.2, 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 check again. The process may have moved from the foreground to the background while it was blocked waiting for input characters.

See also **job control** §B.3.3. A

B.7.1.1.6 Input Processing and Reading Characters

C

B.7.1.1.7 Canonical Mode Input Processing

4.3BSD has a WERASE character that erases the last "word" typed (but not any preceding blanks or tabs). A word is defined as a sequence of non-blank characters, with A A tabs counted as blanks. Like ERASE, WERASE does not erase beyond the beginning of the line. This WERASE feature has not been specified in the standard because it is difficult to define in the international environment. It is only useful for languages where words are delimited by blanks. 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. Thus WERASE would be needless overhead, and has been omitted as superfluous.

B.7.1.1.8 Non-Canonical Mode Input Processing

See c_min and c_time §B.7.1.2.2.

B.7.1.1.9 Writing Characters and Output Processing

B.7.1.1.10 Special Characters

Discussion: The character values for INTR, QUIT, ERASE, KILL, EOF, and EOL, may be changed to suit individual tastes.

B.7.1.1.11 Modem Disconnect

B.7.1.1.12 Closing a Terminal Device File

B.7.1.2 Settable Parameters

B.7.1.2.1 Synopsis

B.7.1.2.2 *termios* Structure

B.7.1.2.3 Input Modes

B.7.1.2.4 Output Modes

B.7.1.2.5 Control Modes

B.7.1.2.6 Local Modes

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

B.7.1.2.7 Special Control Characters

B.7.2 General Terminal Interface Control Functions

B.7.2.1 Get and Set State

B.7.2.2 Line Control Functions

B.7.2.3 Get Distinguished Process Group ID

The *tcgetpgrp*() and *tcsetpgrp*() functions have identical functionality to the 4.2BSD *ioctl*() functions TIOCGPGRP and TIOCSPGRP except for additional security restrictions imposed on *tcsetpgrp*(). The 4.2BSD TIOCSPGRP function allows the caller to associate the terminal with any process group. This allows a user to generate signals from the keyboard that can be sent to any desired process while bypassing the security restrictions imposed by *kill*(). To address this, *tcsetpgrp*() imposes security restrictions similar to *kill*(); the difference is the addition of the saved process group ID. This was added to allow a job control shell to return its controlling terminal to its original process group (the one in effect when the job control shell was executed) regardless of whether the user ID security checks permit it. (Typically the saved process group of a process matches the process group of its parent; but this is not necessarily so.) A job control shell does this before returning control back to its parent when it is terminating or suspending itself. See also *jcsetpgrp*() §B.4.3.2. Note that 4.3BSD closed the 4.2BSD security problem somewhat; it looks for a process whose process ID and process group ID are both equal to the process group supplied to TIOCSPGRP and requires that this process be a descendant of the calling process or that user IDs match. However this still has problems since there may be processes which belong to the specified process group, but which are not the process group leader. This is actually a frequent occurance since *csh* makes the first process in a pipeline be the process group leader and this process is usually the first to terminate. See also **job control** §B.3.3. A

B.7.2.4 Set Distinguished Process Group ID

B.8 C Language Library

When the *ANSI/X3.159-198x Programming Language C Standard* is adopted, it will be the basis for a C language binding to POSIX. In the interim, the routines in this chapter are left unstandardized, but are defined by common usage and traditional implementations. Common usage may also be derived by such historical publications as *The C Programming Language*, by Kernighan and Ritchie, listed in **Bibliographic Notes** §B.11.

The null set of supported languages is allowed.

B.8.1 Referenced C Language Routines

B.8.1.1 Extensions to *asctime*() Function

System V uses the TZ environment variable to set some information about time. It has the form (spaces inserted for clarity):

std offset dst

where the first three characters (*std*) are the name of the standard time zone, the digits which follow (*offset*) are the hours West of Greenwich (or, if preceded by "–", East), and the next three characters (*dst*) are the name of the summer time zone. Both *std* and *offset* are required; if *dst* is missing, summer time does not apply.

Currently, the UNIX system **localtime** function translates a number of seconds since The Epoch §2.3 into a detailed breakdown of that time. This breakdown includes:

- Time of day: Hours, minutes, and seconds.
- Day of the month, month of the year, and the year.
- Day of the week and day of the year (Julian day).
- Whether or not summer (daylight saving) time is in effect.

It is first and last items that present a nasty problem: The time of the day depends on whether or not summer time is in effect. Whether or not summer time is in effect depends on the locale and date.

Currently the UNIX system has built into it only the United States federal law for the years 1970 to 1986. The U.S. law was changed for 1987 and subsequent years, so much UNIX system software is now "broken." Actually, 4.2BSD includes time zone rules in a file that does take Europe and Australia into account. There are some errors and limitations with this method. And if the system is outside the United States, that same UNIX system software has always been broken.

The challenge is to fix the existing built-in rules for the new U.S. law and, in the process, extend **localtime** so that non-U.S. locales won't suffer from Yankee daylight saving time. Fixing the built-in rule is straightforward. Extending **localtime** is less so.

This proposal extends the existing **TZ** environment variable (which names the locale's time zone) to also include a rule for when to use standard time and when to use summer time. Southern hemisphere time zones are supported by allowing the first *rule date* (change to summer time) to be later in the year than the second *rule date* (change to standard time).

The proposal accommodates the "floating day" rules (for example "last Sunday in October") used in the U.S. and Canada (and the European Economic Community for the last several years). In theory, **TZ** only has to be set once and then never touched again unless the law is changed.

Julian dates are proposed with two syntaxes, one zero based, the other one based. They are here for historical reasons. The one based counting (*J*) is used more commonly in Europe (and on calendars people may use for reference). The zero based counting (*n*) is used currently in some implementations and should be kept for historical reasons as well as being the only way to specify Leap day.

It is expected the leading slash followed by some bytes as either the entire **TZ** string or as the *rule* will enable systems to have time zone information included in a file (as 4.2BSD systems currently do) or use the bytes as an index into a database. The implementors have the option as to how these bytes are interpreted. Allowing the implementors to specify either the entire time zone or the *rule* makes the proposal capable of describing the complete history for a multitude of locales. This proposal speculates that very few programs actually need to be historically accurate as long as the relative timing of two events is preserved. But, for the probably few programs that do desire such accuracy, the */bytes* method is provided.

Summer time is governed by both locale and date. This proposal only handles the locale dependency. Using an implementation defined file format for either the entire **TZ** variable or to specify the *rules* for a particular time zone is allowed as a means by which both the locale and date dependency can be handled.

Since current implementations do not examine **TZ** beyond the assumed end of *dst*, it is possible to literally extend **TZ** and break very little existing software. Since much of the software doesn't work anyway outside the U.S. time zones, minor changes to **TZ** (such as extending *offset* to be *hh*:*mm* — as long as the colon and minutes, *:mm*, are optional) will have little impact.

B.8.1.2 Extensions to *setlocale*() Function

Recently, the ANSI X3J11 subcommittee issued a draft proposal for the C Programming Language. In addition to many changes to the language, the proposal defines a collection of interfaces to support internationalization. One of the most significant aspects of these interfaces is a facility to set and query the *international environment*. The international environment is a repository of information that affects the behavior of certain functionality, namely:

- Character Handling
- String Handling (i.e., collating)
- Date/Time Formatting
- Numeric Editing

The *setlocale*() function provides the application developer with the ability to set all or portions, called *categories*, of the international environment. These categories correspond to the areas of functionality, mentioned above. The syntax for setlocale is the following:

```
char     *setlocale (category, locale)
int      category;
char     *locale
```

Where *category* is the name of one of four categories, namely:

```
LC_CTYPE
LC_COLLATE
LC_TIME
LC_NUMERIC
```

In addition, a special value, called LC_ALL, directs *setlocale*() to set all categories.

The *locale* argument is a character string that points to a specific setting for the international environment, or locale. There are three preset values for the locale argument, namely:

C	Specifies the minimal environment for C translation. If setlocale is not invoked, the "C" locale is the default.
""	Specifies an implementation-defined native environment.
NULL	Used to direct *setlocale*() to query the current international environment and return the name of the locale.

This section describes the behavior of an implementation of *setlocale*() and its use of environment variables in controlling this behavior on POSIX-based systems. There are two primary uses of *setlocale*():

- Querying the international environment to find out what it is set to,
- Setting the international environment, or *locale*, to a specific value.

The following sub-sections will describe the behavior of *setlocale*() in these two ares. Since it is difficult to describe the behavior in words, examples will be used to illustrate the behavior of specific uses.

To query the international environment, *setlocale*() is invoked with a specific category and the null pointer as the locale. The null pointer is a special directive to *setlocale*() that tells it to query rather than set the international environment. Below is the syntax for using *setlocale*() to query the name of the international environment:

setlocale() returns the string corresponding to the current international environment. This value may be used by a subsequent call to *setlocale*() to reset the international environment to this value. However, it should be noted that the return value from *setlocale*() is a pointer to a static area within the function and is not guaranteed to remain unchanged (i.e., it may be modified by a subsequent call to *setlocale*()). Therefore, if the purpose of calling *setlocale*() is to save the value of the current international environment so it can be changed and reset back later, the return value should be copied to a character array in the calling program.

There are three ways to set the international environment with *setlocale*():

setlocale(*category*, *string***)**

This usage will set a specific *category* in the international environment to a specific value corresponding to the value of the *string*. A specific example is provided below:

```
setlocale(LC_ALL, "Fr_FR.8859");
```

In this example, all categories of the international environment will be set to the locale corresponding to the string **"Fr_FR.8859"**, or the french language as spoken in France using the ISO 8859/1 code set.

If the string does not correspond to a valid locale, *setlocale*() will return a null pointer and the international environment is not changed. Otherwise, setlocale will return the name of the locale just set.

setlocale(*category*, **"C")**

The ANSI X3J11 draft proposal states that one locale must exist on all conforming implementations. The name of the locale is **"C"**, and corresponds to a minimal international environment needed to support the C programming language.

setlocale(*category*, **"")**

This will set a specific category to an implementation-defined default. For POSIX-based systems, this corresponds to the value of the environment variables.

B.8.2 FILE-Type C Language Functions

B.8.2.1 Map a Stream Pointer to a File Descriptor

Without some specification of which file descriptors are associated with these streams, it is impossible for an application to set up the streams for another application it starts with *fork*() §3.1.1 and *exec* §3.1.2. In particular, it would not be possible to write a portable version of the `sh` command processor (although there may be other constraints that would prevent that portability).

Note that this standard permits an implementation to associate file descriptors other than 0, 1, and 2 with *stdin*, *stdout*, and *stderr*.

B.8.2.2 Open a Stream on a File Descriptor

The file descriptor may have been obtained from *open*() §5.3.1, *creat*() §5.3.2, *pipe*() §6.1.1, *dup*() §6.2.1, *fcntl*() §6.5.2, or inherited through *fork*() §3.1.1 or *exec* §3.1.2, or perhaps obtained by implementation-dependent means, such as the 4.3BSD *socket*() call.

The meanings of the *type* arguments of *fdopen* and *fopen* differ. With *fdopen*, open for write ("w" or "w+") does not truncate and append ("a" or "a+") cannot create for writing. There is no need for "b" in the format due to the equivalence of binary and text files in POSIX. See Text vs. binary file modes §B.1.4.

B.8.3 Other C Language Functions

B.8.3.1 Non-Local Jumps

X3J11 specifies various restrictions on the usage of the *setjump*() macro in order to permit implementors to recognize the name in the compiler and not implement an actual function. These same restrictions apply to the *sigsetjmp*() macro.

The names of these functions were changed to *sigsetjmp*() and *siglongjmp*(). This avoided conflict with the C Standard *setjmp*() and *longjmp*(), which do not have the same behavior in regards to signal masks.

There are processors that cannot easily support these calls, but the Working Group did not consider that a sufficient reason not to include them.

The distinction between *setjmp*()/*longjmp*() and *sigsetjmp*()/*siglongjmp*() is only significant for programs which use the *sigaction*(), *sigprocmask*(), or *sigsuspend*() functions.

BSD systems provide functions named *_setjmp*() and *_longjmp*() which, together with *setjmp*()/*longjmp*(), save and restore signal masks. While many other systems provide versions of these functions that do not, the Working Group decided not to specify the relation of these functions to signal masks and to define a new set of functions instead.

B.8.3.2 Specify Signal Handling

The *sigaction*() §3.3.4 was introduced in order to provide an interface for reliable signal handling (see **Singals** §B.3.3). The *signal*() function is included in this document because *signal*() is defined in the *ANSI/X3.159-198x Programming Language C Standard*. However, it is recommended that POSIX applications use only the *sigaction*() interface, due to the potential unreliability and lack of consistency among existing *signal*() implementations. Portable library routines often need to install a signal catching function and then restore the signal to its original state. The function *sigaction*() should always work correctly for this purpose, regardless of what the rest of the program does. The *signal*() function may not work correctly if other parts of the program use *sigaction*().

It is the intention of the Working Group that *signal*() be implementable as a library routine using *sigaction*().

B.9 System Databases

At one time, this chapter was entitled **Passwords**, but this title was changed as all references to a "password file" were changed to refer to a "user database."

B.9.1 System Databases

There are no references in the standard to a **passwd file** §B.2.3 or a **group file** §B.2.3 and there is no requirement that the *group* or *passwd* databases be kept in ASCII files. 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 encoded password fields were deleted from both the *passwd* and *group* databases in order to meet the requirements of the US Government NBS Password FIPS (and FIPS concerns in general).

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

B.9.2 Database Access

B.9.2.1 Group Database Access

B.9.2.2 User Database Access

B.10 Data Interchange Format

B.10.1 Archive/Interchange File Format

There are three areas of interest associated with file interchange:

Media There are other existing standards that define the media used for data interchange.

User Interface This rightfully should be in the IEEE Std 1003.2 standard.

Format of the Data None of the P1003 Working Groups address topics that match this area. The Working Group feels that this area is closest to the types of things that should be in the IEEE Std 1003.1 document, as the level of that document most closely matches the level of data required.

There appear to be two programs in wide use today, `tar` and `cpio`. There are large camps of supporters for each program. Four options were considered for the standard:

1. Make both formats optional. This was considered unacceptable because it does not allow any portable method for data interchange.
2. Require one format.
3. Require one format with the other optional.
4. Require both formats.

This issue is not yet resolved. In the September 1987 meeting, the `cpio` format was approved for inclusion in the standard as the data interchange format. The Extended `tar` Format was placed into Appendix D to solicit Balloting Group opinions on this issue.

There are a number of concerns about defining extensions that are known to be required by existing implementations. Failure to specify a consistent method to implement these extensions will severely limit portability of the program and, more importantly, will create severe confusion if these extensions are later standardized.

Two of these extensions that the Working Group felt should be documented are symbolic links, that were defined by 4.2BSD and 4.3BSD systems, and high performance (or contiguous) files, that exist in a number of implementations and are now being considered for the 1003.4 standard.

By defining these extensions, implementors are able to recognize these features and take appropriate implementation defined actions for these files. For example, a high performance file could be converted to a regular file if the system didn't support high performance files; symbolic links might be replaced by normal hard links.

The Working Group has held to the policy of not defining user interfaces to utilities by avoiding any description of a `tar` or `cpio` command. The behavior of the former command was described in some detail in previous drafts.

The possibilities for transportable media include, but are not limited to,

1. 1/2 inch magnetic tape, 9 track, 1600 BPI
2. 1/2 inch magnetic tape, 9 track, 6250 BPI
3. Qic-11, 1/4 inch streamer tape
4. Qic-24, 1/4 inch streamer tape
5. 5.25 inch floppies, 8 512-byte sectors/track, 96 TPI
6. 5.25 inch floppies, 8 512-byte sectors/track, 48 TPI
7. IBM 3480 cartridges.

Specification of such media was considered part of the scope of the Trial Use Standard, but has been excluded from the Full Use Standard.

The utilities are not restricted to work only with *transportable* media: existing related utilities are often used to transport data from one place to another in the file hierarchy.

The format is included to provide an implementation independent way to move files from one system to another and also to provide a way for a user to save data on a transportable medium to be restored at a later date. Unfortunately, these two goals can contradict each other as system security problems are easy to find in tape systems if they are not protected. Thus the strict requirements about how the mechanism to copy files shall react when operated by both privileged and nonprivileged users. The general concept is that a privileged (using the ISUID bit in the file's mode with the owner UID of the file set to super-user) version of the utility can be used as a save/restore scheme, but a nonprivileged version is used to interpret media from a different system without compromising system security.

Regardless of the archive format used, guidelines should be observed when writing tapes to be read on other systems. Assuming the target system is POSIX compliant, archives created should use only use definitions found in POSIX (e.g., file types, minimum values as found in Chapter 2) and should only use relative pathnames (i.e., no leading /).

B.10.1.1 `cpio` Archive Format

The model for this format is the existing System V `cpio -c` data interchange format. This models documents the portable version of `cpio` format and *not* the binary version. It has the flexibility to transfer data of any type described within the POSIX standard, yet is extensible to transfer data types specific to extensions beyond POSIX (e.g., symbolic links or contiguous files). Because it describes existing practice, there is no question of maintaining upward compatibility.

This section does not standardize behavior for the utility when the file type is not understood or supported. It is useful for the utility to report to the user whatever action is taken in this case, though the standard neither requires nor recommends this.

B.10.1.1.1 Header

There has been some concern that the size of the *c_ino* field of the header is too small to handle those systems which have very large i-node numbers. However, the *c_ino* field in the header is used strictly as a hard link resolution mechanism for archives. It is not necessarily the same value as the i-node number of the file in the location that file is extracted from.

B.10.1.1.2 File Name

For most current implementations of the `cpio` utility, {PATH_MAX} bytes can be used to describe the pathname without the addition of any other header fields (the null byte would be included in this count). {PATH_MAX} is the minimum value for pathname size, documented as 256 bytes in Chapter 2 of the standard. However, an implementation may use *c_namesize* to determine the exact length of the pathname. With the current description of the `cpio` header, this pathname size can be as large as a number which is described in six octal bytes.

B.10.1.1.3 File Data

B.10.1.1.4 Special Entries

These are provided to maintain backward compatibility.

B.10.1.1.5 `cpio` Values

Three values are documented under the *c_mode* field values to provide for extensibility for known file types:

110000 — Suggested symbolic name—ISCTG; reserved for contiguous files. The implementation may treat the rest of the information for this archive like a regular file. If this file type is undefined, the implementation may create the file as a regular file.

120000 — Suggested symbolic name—ISLNK; reserved for files with symbolic links. The implementation may store the link name within the data portion of the file. If this type is undefined, the implementation may not know how to link this file or be able to understand the data section. The implementation may decide to

	ignore this file type and output a warning message.
140000	Suggested symbolic name—ISSOCK; reserved for sockets. If this type is undefined on the target system, the implementation may decide to ignore this file type and output a warning message.

This provides for extensibility of the **cpio** format while allowing for the ability to read old archives. Files of an unknown type may be read as "regular files" on some implementations.

B.10.1.2 Multiple Volumes

Multi-volume archives have been introduced in a manner that has become a *de facto* standard in many implementations. Though it is not required by POSIX classical implementations of the format-reading and -creating utility, upon reading logical end-of-file, check to see if an error channel is open to a controlling terminal. The utility then produces a message requesting a new medium to be made available. The utility waits for new medium to be made available by attempting to read a message to restart from the controlling terminal. In all cases, the communication with the controlling terminal is in an implementation defined manner.

The section **Multiple Volumes** §10.1.2 is intended to handle the issue of multiple volume archives. Since the end-of-medium and transition between media are not properly part of this standard, the transition is described in terms of files.

The intent is that files will be read serially until the end-of-archive indication is encountered, and that file or media change will be handled by the utilities in an implementation defined manner.

Note that there was an issue with the representation of this on magnetic tape, and the standard is intended to be interpreted such that each byte of the format is represented on the media exactly once. In some current implementations, it is not deterministic whether encountering the end-of-medium reflector foil on magnetic tape during a write will yield an error during a subsequent *read*() of that record, or if that record is actually recorded on the tape. It is also possible that *read*() will encounter the end-of-medium when end-of-medium was not encountered when the data was written. This has to do with conditions where the end of [magnetic] record is in such a position that the reflector foil is on the verge of being detected by the sensor and is detected during one operation and not on a later one, or vice-versa.

An implementation of the format-creating utility must assure when it writes a record that the data appears on the tape exactly once. This implies that the program and the tape driver work in concert. An implementation of the format-reading utility must assure that an error in a boundary condition described above will not cause loss of data.

The general consensus was that the following would be considered as correct operation of a tape driver when end-of-medium is detected:

During writing, either:

1. The record where the relector spot was deleted is backspaced over by the driver so that the trailing tape mark that will be written on *close*() will overwrite.

   Writing the tape mark should not yield an end-of-medium condition.

2. Or, the condition is reported as an error on the *write*() following the one where the end-of-medium is detected (the one where the end-of-medium is actually detected completing successfully). No data will be actually transferred on the *write*() reporting the error. The subsequent *close*() would *write*() a tape mark following the last record actually written.

   Writing the tape mark, and writing any subsequent records, should not yield any end-of-medium conditions.

(The latter behavior permits the implementation of ANSI standard labels because several records (the trailer records) can be written after the end-of-medium indications. It also permits dealing with, for example, COBOL "ON" statements.)

During reading:

The end-of-medium indicator is simply ignored, presuming that a tape mark (end-of-file) will be recorded on the magnetic medium, and the reflector foil was advisory only to the *write*().

Systems where these conditions are not met by the tape driver should assure that the format-creating and -reading utilities assure proper representation and interpretations of the files on the media, in a way consistent with the above recommendations.

The typical failures on systems that do not meet the above conditions are either:

1. To leave the record written when the end-of-medium is encountered on the tape, but to report that it was not written. The format-creating utility would then rewrite it, and then the format-reading utility could see the record twice if the end-of-medium is not sensed during the read operations.

2. Or, the *write*() occurs uneventfully, but the *read*() senses the error and does not actually see the data, causing a record to be omitted.

Nothing in this standard requires that end-of-medium be determined by anything on the medium itself (for example, a predetermined maximum size would be an acceptable solution for the format creating utility). The format-reading utility must be able to *read*() tapes written by machines that do use the whole medium, however.

On media where end-of-medium and end-of-file are reliably coincident, such as disks, end-of-medium and end-of-file can be treated as synonyms.

Note that partial physical records (corresponding to a single *write*()) can be written on some media, but that only full physical records will actually be written to magnetic tape, given the way the tape operates.

B.10.1.3 Extended `tar` Format

This section was originally in the body of the Trial Use Standard but was moved to Appendix D for the Full Use Ballot.

The original model for this facility is the 4.3BSD or Version 7 `tar` program and format, but the format given here is an extension of the traditional `tar` format. The name USTAR was adopted to reflect this.

This description reflects numerous enhancements over previous versions. The goal of these changes was not only to provide the functional enhancements desired, but to retain compatibility between new and old versions. This compatibility has been retained. Archives written using the old archive format are compatible with the new format. Archives written using this new format may be read by applications designed to use the old format as long as the functional enhancements provided here are not used. This means the user is limited to archiving only regular type files and nonsymbolic links to such files.

If a utility reads an archive that contains file types that the utility either does not understand or does not support (such as symbolic links or contiguous files), it is useful for the utility to report whatever action it takes to the user, though the standard neither requires nor recommends this.

Implementors should be aware that the previous file format did not include a mechanism to archive directory type files. For this reason, the convention of using a file name ending with slash was adopted to specify a directory on the archive.

Note that, NAMSIZ plus PFXSIZ have been set to meet the minimum requirements for {PATH_MAX}. If a pathname is less than NAMSIZ−1 characters and therefore fits within the *name* field, it is recommended that the pathname be stored there without the use of the *prefix* field. Although the value of NAMSIZ is known to be less than {PATH_MAX}, the value was not changed in this version of the archive file format to retain backward compatibility and instead the *prefix* was introduced. Also because of the earlier version of the format, there is no way to remove the limitation on the *linkname* field being set to NAMSIZ.

B.11 Bibliographic Notes

There are far more related papers and books than are mentioned here, and some of them may be as good or better.

B.11.1 Related Standards

The standard assumes that any terms not defined in Chapter 2 are defined in the *IEEE Standard Dictionary of Electrical and Electronics Terms*, IEEE Std 100-1977.

The *1984 /usr/group Standard* may be ordered from

/usr/group Standards Committee
4655 Old Ironsides Drive, Suite 200
Santa Clara, California 95054
(408)986-8840

The basic historical reference on the C language is

- Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

The *ANSI/X3.159-198x Programming Language C Standard* may be obtained from

Global Press
2625 Hickory St.
P.O. Box 2504
Santa Anna, CA 92707-3783
U.S.A.

800-854-7179
+1-714-540-9870 (from outside the U.S., ask for extension 245.)
TELEX 692 373

The *X/OPEN Portability Guide* is published by

Elsevier Science Publishers B.V.
Book Order Department
P.O. Box 1991
1000 BZ Amsterdam
The Netherlands

and is distributed in the United States and Canada by

Elsevier Science Publishing Company, Inc.
52 Vanderbilt Avenue
New York, NY 10017
U.S.A.

There are five volumes, of which Volume 2 is the most relevant to the present standard.

B.11.2 Historical Implementations

A principal ancestor of all the historical implementations is the Multics System

- Organick, Elliot I., *The Multics System: An Examination of Its Structure,* The MIT Press, Cambridge, MA (1975).

The most basic and influential paper on historical implementations is

- Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System," *Bell System Technical Journal* 57(6 Part 2) pp. 1905-1929 American Telephone and Telegraph Company, (July-August 1978). This is a revised version and describes Version 7.
- Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System," *Commun. ACM* 7(7) pp. 365-375 Association for Computing Machinery, (July 1974). This is the original paper, which describes Version 6.

The Version 7 manual is

- AT&T, *UNIX Time Sharing System: UNIX Programmer's Manual, Seventh Edition,* Bell Telephone Laboratories, Inc., Murray Hill, New Jersey (January, 1979).

Dennis Ritchie has also done several papers on the history and evolution of the system

- Ritchie, Dennis, "The Evolution of the UNIX Time-sharing System," *AT&T Bell Laboratories Technical Journal* 63(8) pp. 1577-1593 American Telephone and Telegraph Company, (October 1984).
- Ritchie, Dennis M., "Reflections on Software Research," *Commun. ACM* ?(?) p. ? Association for Computing Machinery, (1984). ACM Turing Award Lecture
- Ritchie, Dennis M., "Unix: A Dialectic," *USENIX Association Conference Proceedings*, pp. 29-34 USENIX Association, P.O. Box 2299, Berkeley, CA 94710, (21-23 January 1987).

Important collections of papers on the system may be found in

- BSTJ, "UNIX Time-Sharing System," *Bell System Technical Journal* 57(6 Part 2)American Telephone and Telegraph Company, (July-August 1978).
- BLTJ, "The UNIX System," *AT&T Bell Laboratories Technical Journal* 63(8)American Telephone and Telegraph Company, (October 1984).

The System III manual is

- AT&T, *UNIX System III Programmer's Manual,* Western Electric Company, Inc., Greensboro, N.C. (October, 1981).

The *SVID*

- AT&T, *System V Interface Definition, Issue 2*, AT&T (1986).

may be ordered from

AT&T Customer Information Center
Attn: Customer Service Representative
P.O. Box 19901
Indianapolis, IN 46219
U.S.A.

800-432-6600 (Inside U.S.A.)
800-255-1242 (Inside Canada)
317-352-8557 (Outside U.S.A. and Canada)

using the following Select Codes:

320-011	Volume I
320-012	Volume II
320-013	Volume III
307-131	all three volumes

The implementation of System V is described in

- Bach, Maurice J., *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, New Jersey (1986).

The 4.3BSD manual

- UCB-CSRG,, *4.3 Berkeley Software Distribution, Virtual VAX-11 Version*, The Regents of the University of California, Berkeley, California (April 1986).

is printed by the USENIX Association, and their members may order from them:

USENIX Association
P.O. Box 2299
Berkeley, CA 94710
415-528-8649

The implementation of the kernel of 4.3BSD is described in

- Quarterman, John S., Silberschatz, Abraham, and Peterson, James L., "4.2BSD and 4.3BSD as Examples of the UNIX System," *ACM Computing Surveys* 17(4) pp. 379-418 Association for Computing Machinery, (December 1985).

- Leffler, Samuel J., McKusick, Marshall Kirk, Karels, Michael J., Quarterman, John S., and Stettner, Armando, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, Massachusetts (1988).

B.11.3 Historical Application Programming Tutorials

A useful tutorial on programming in the C language is

- Harbison, Samuel P. and Steele, Guy L., *C: A Reference Manual,* Prentice-Hall, Englewood Cliffs, New Jersey (1987).

A highly regarded book, though not one for beginners, is

- Kernighan, Brian W. and Pike, Rob, *The UNIX Programming Environment,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1984).

One more oriented towards Berkeley systems is

- McGilton, Henry and Morgan, Rachel, *Introducing the UNIX System,* McGraw-Hill (BYTE Books), New York (1983).

and a more recent one is

- Rochkind, Marc J., *Advanced UNIX Programming,* Prentice-Hall, Englewood Cliffs, New Jersey (1985).

C. Comparison to *System V Interface Definition*

The *System V Interface Definition* (*SVID*) defines the external characteristics (externally visible interfaces and behavior) common to all System V environments. When it was first published in 1984, it differed in small ways with the *1984 /usr/group Standard*, and those differences were listed in Issue 1 of the *SVID*. This appendix lists the differences between Issue 2 of the *SVID* (Volumes 1-3) and the IEEE Std 1003.1. Unless otherwise noted, all differences are compared to the BASE definition of the *SVID*. Overall differences are described first and then differences in specific functions are described. All known differences in defined functionality are listed although some may be of minor importance.

In most cases, on a specific point of difference, both IEEE Std 1003.1 and *SVID* definitions are presented. In other cases, particularly when one document includes a point that the other does not, only the statement from that document is characterized.

Numbers in parentheses below, such as (2.3) or (3.2.2.2), refer to sections in IEEE Std 1003.1.

C.1 Overall Contents

C.1.1 Operating System Primitives

Functions included only in

1003.1: *mkfifo*(), *getgroups*(), *rename*(), *pathconf*(), *fpathconf*(), *sysconf*().

mkfifo(), *pathconf*(), and *sysconf*() are new functions.

In System V, FIFO files are made with the *mknod*() function.

The optional *getgroups*() function is not included in the *SVID*,

The *rename*() function is not included in the *SVID*.

SVID: *ioctl*(), *mknod*(), *mount*(), *umount*(), *pclose*(), *popen*(), *stime*(), *sync*(), *ulimit*(), *ustat*().

The *SVID* defines these ten additional functions and requires them to be supported by any System V environment.

C.1.2 Library Routines

Functions described only in:

1003.1: Eleven routines are included in 1003.1 that are not found in the Base System definition in the *SVID*, but are found in the Software Development Extension. These include five routines that access the group database (/etc/**group** in *SVID*): *endgrent*(), *getgrent*(), *setgrent*(), *getgrid*(), *getgrnam*(); five routines that access the passwd database (/etc/passwd in *SVID*): *endpwent*(), *getpwent*(), *setpwent*(), *getpwnam*(), *getpwuid*(); one routines to return user login names, *getlogin*(). One routine is included in 1003.1 that is not in the Software Development Extension in the *SVID*: *cuserid*().

SVID: The *SVID* defines approximately 150 additional routines many of which are covered in the *ANSI/X3.159-198x Programming Language C Standard*. and are included in 1003.1 by reference (8.1). Any differences between the *SVID* definitions and the *ANSI/X3.159-198x Programming Language C Standard* definitions are not covered in this appendix. These include math routines, memory allocation, non-local jumps, data conversion and encoding, *stdio* routines, string and character handling, sorting, regular expression matching, search routines and some others.

C.1.3 Special Files

SVID: Three special device files are required by the *SVID*,

/dev/console	system console interface
/dev/null	the null file
/dev/tty	controlling terminal interface.

C.1.4 Minimal Directory Tree Structure

SVID: Specifies a minimal directory tree structure comprising **/bin**, **/dev**, **/etc**, **/tmp**, **/usr/bin**, and **/usr/tmp**.

C.1.5 Multiple Groups

1003.1: Defines supplemental groups as an optional feature ({NGROUPS_MAX} may be zero). This feature affects several components of the standard.

C.1.6 Job Control

1003.1: Defines job control as an optional feature. None of the functions detailed here are included unless the Job Control Option is present. This feature affects several components of the standard: four functions (*jcsetpgrp*(), *tcgetpgrp*(), *tcsetpgrp*(), and *wait2*()) and a header file (<wait.h>) have been added to the standard. In addition, the *signal*() definition was affected and other signals were added.

SVID: Does not include the Job Control option.

C.1.7 Enhanced Signals

1003.1: (3.3.3) Extends the signal handling functions to include a set of functions that manage sets of signals. The functions *siginitset*(), *sigfillset*(), *sigaddset*(), *sigdelset*(), *sigismember*(), *sigaction*(), *sigprocmask*(), *sigpending*(), *sigsuspend*() were added to the standard. The structure definition *sigaction* was added to the header file **<signal.h>**.

1003.1: Specifies that the signal mask is conditionally saved and restored by the *sigsetjmp*() and *siglongjmp*() functions.

SVID: Volume 3 added functions to support an extended form of signal handling. The functions *sigset*, *sighold*, *sigrelse*, and *sigignore* were added. All functions takes a single signal number of type **int**. The *sigset* function takes an additional parameter which is one of four values: SIG_DFL, SIG_IGN, SIG_HOLD, or an *address* of a signal-catching function.

C.1.8 Configurable System Variables

1003.1: Three new functions, *fpathconf*(), *pathconf*(), and *sysconf*(), were added to the system configuration variables.

C.1.9 Terminal I/O

The comparison described here is between *termios* from 1003.1 and *termio* from *SVID*.

1003.1: (7.1) Specifies a set of functions to manipulate a terminal.

SVID: Specifies a set of *ioctl* commands to manipulate a terminal.

C.2 Specific Differences

C.2.1 Error Numbers

1003.1: (2.5) Includes the additional errors

ENAMETOOLONG	filename too long
ENOTEMPTY	directory not empty

SVID: Includes the additional error

ENOTBLK	block device required
ETXTBSY	program text file busy

SVID: Volume 3 of the *SVID* specifies as a future direction, that in the case of a path-name argument exceeding {PATH_MAX}, the error returned would change to follow the direction of the 1003.1 standard. Volume 3 currently specifies ENOENT as the error returned.

C.2.2 General Terms

1003.1: (2.4) **pathname searches**—As a special case, in the **root directory**, "dot-dot" *may* refer back to **root directory** itself.

SVID: **directory**—The root directory, which is the top-most node of the hierarchy, has itself as its parent directory.

C.2.3 Data Types

1003.1: The defined type **time_t** is time measured in seconds and **clock_t** is time measured in {CLK_TCK}ths of a second. (2.6)

SVID: The defined type **time_t** is time measured in either {CLK_TCK}ths of a second (*times*()) or in seconds (*stat*()). The type **clock_t** is not defined in *SVID*.

1003.1: The defined type **uid_t** is used to represent user and group IDs. As a result, differences in synopses exist in the following functions: *getuid*(), *geteuid*(), *getgid*(), *getegid*(), *setuid*(), *setgid*(), **<sys/stat.h>**, and *chown*().

1003.1: The defined type **mode_t** is used to represent file modes. As a result, differences in synopses exist in the following functions: *creat*(), *umask*(), *mkdir*(), *open*(), **<sys/stat.h>**, and *chmod*.

C.2.4 Environment Variables

1003.1: (2.7) Defines additional variables that may be defined: PS1, PS2, IFS, MAIL, SHELL, LOGNAME, LC_CTYPE, LC_COLLATE, LC_TIME, LC_TIME, and LC_NUMERIC.

C.2.5 fork()

1003.1: (3.1.1.2) Lists attributes *not* inherited by the child process and specifies that all other attributes defined by the standard shall be inherited. Implementations may add characteristics that are or are not inherited.

SVID: Lists attributes that must be inherited as well as those not inherited by the child process.

C.2.6 exec()

SVID: When a C program is executed, it is called as follows

```
main (argc, argv, envp)
int argc;
char **argv, **envp;
```

In 1003.1, (3.1.2.2) the third argument, is not specified. B

SVID: The effective user ID and group ID of the new process are saved for use by *setuid*(). In 1003.1, (3.1.2.2) this is optional.

SVID: Specifies that the new process additionally inherits the terminal group id and file-size limit of the calling process. 9

C.2.7 wait()

1003.1: (3.2.1.2) If the child process terminated due to a signal that was not caught, the low order *6 bits of status* will contain the signal number.

SVID: If the child process terminated due to a signal that was not caught, the low order *7 bits of status* will contain the signal number.

1003.1: Additionally allows *wait*() to return due to an implementation-defined change in the status of a child process. A

C.2.8 _exit()

1003.1: (3.2.2.2) If the calling process is the process group leader, SIGHUP *may be* sent to each process with a process group ID equal to the calling process.

SVID: If the calling process is a process group leader *and is associated with a controlling terminal,* SIGHUP *is* sent to each process with a process group ID equal to that of the calling process. 8

1003.1: If a child process is stopped under job control, it will be sent both SIGHUP and SIGCONT. 8

C.2.9 <signal.h>

1003.1: (3.3.1.2) The additional signal SIGSEGV is defined.

SVID: The signal SIGSEGV is not on the list of signals that applications should know about and the *SVID* warns that its meaning is implementation-dependent.

SVID: The additional signal SIGSYS, bad argument to system call, is defined. This signal is not in 1003.1.

SVID: The signal SIGABRT defined in 1003.1 is indicated in *SVID* Volume 3.

1003.1 The signals SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, and SIGCONT are optional based on the presence of the Job Control Option.

C.2.10 kill()

SVID: Specifies that an error is returned if the arguments *sig* is SIGKILL and *pid* is a special system process.

1003.1 (3.3.2.2) Specifies that if the signal is being sent to all processes, the sender *may* be excluded. 1003.1 also specifies that if both {_POSIX_KILL_SAVED} and {_POSIX_SAVED_IDS} are defined, the saved set-user-ID of the receiving process shall be checked in place of its effective user ID.

C.2.11 signal()

1003.1: (3.3.8.2) A call to *signal*() shall cancel a pending signal if the *func* parameter is SIG_IGN, and may cancel pending signals, except for a pending SIGKILL signal.

SVID: A call to *signal*() cancels a pending signal *of type sig* except for a pending SIGKILL signal. (Note that only a pending signal of the same type for which signal was just called is affected.)

C.2.12 times()

1003.1: (4.5.2.2) Specifies the members of the *tms* structure as type **clock_t**.

SVID: Specifies the members of the *tms* structure as type **time_t**.

C.2.13 open()

1003.1: (5.3.1.2) When a file is created with the O_CREAT flag, 1003.1 specifies that the file's group ID shall be set to either the process's effective group ID *or* to the group ID of the directory in which the file is being created.

SVID: Specifies that when a file is created with the O_CREAT flag, the file's group ID is set to the process's effective group ID.

1003.1: Specifies the flag O_NONBLOCK.

SVID: Specifies the flag O_NDELAY.

SVID: Specifies two additional error conditions.

ENXIO	The named file is a character special or block special file and the device associated with the special file does not exist.
ETXTBSY	The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write.

C.2.14 unlink()

SVID: Specifies the additional error condition

ETXTBSY	The entry to be unlinked is the last link to a pure procedure file that is being executed.

C.2.15 rmdir()

1003.1: (5.5.2.1) Specifies that an implementation can return either EEXIST or ENOTEMPTY if the directory being removed contains files.

SVID: Specifies that an implementation shall return EEXIST if the directory being removed contains files.

C.2.16 <sys/stat.h>

1003.1: Recommends the S_ISUID and the S_ISGID bits be cleared on every *write*.

C.2.17 access()

1003.1: Specifies the optional error condition

EINVAL	Invalid value for *amode*.

SVID: Specifies the additional error condition

ETXTBSY	Write access requested for a pure procedure file that is being executed.

C.2.18 chown()

1003.1: (5.6.5.4) Specifies the optional error condition

EINVAL	The owner or group ID supplied is outside the range of 0 to {UID_MAX}, inclusive.

C.2.19 utime()

1003.1: (5.6.6.2) Specifies the inclusion of **<utime.h>** which defines the *utimbuf* structure.

SVID: The *utimbuf* structure must be defined by the user.

C.2.20 close()

1003.1: (6.3.1.1) Specifies the additional error condition

EINTR	The *close* function was terminated prematurely by a signal.

C.2.21 read()

1003.1: (6.4.1.4) Specifies that an error will be returned and *errno* set to EAGAIN if no-delay (O_NONBLOCK) mode is in effect and the process would be delayed in reading.

***SVID*:** *SVID* Volume 3 specifies read will return 0 in the no-delay (O_NDELAY) case. The change to return EAGAIN is listed as a future direction.

***SVID*:** Specifies the additional errors

- EIO — A physical I/O error has occurred
- ENXIO — The device associated with the file descriptor is a block-special or character-special file and the value of the file pointer is out of range.

C.2.22 write()

1003.1: (6.4.2.4) Specifies the additional error condition:

- EAGAIN — O_NONBLOCK is set and the process would be delayed in the *write*() operation

***SVID*:** Specifies the additional errors

- EIO — A physical I/O error has occurred
- ENXIO — The device associated with the file descriptor is a block-special or character-special file and the value of the file pointer is out of range.

***SVID*:** Specifies that in the O_NDELAY case, if the write request doesn't transfer data, 0 is returned.

C.2.23 <fcntl.h>

1003.1: (6.5.1.2) Specifies the symbolic name of the no-delay flag to be O_NONBLOCK.

SVID: Specifies the symbolic name of the no-delay flag to be O_NDELAY.

C.2.24 fcntl()

1003.1: (6.5.2). Specifies the additional error condition

EINTR The *fcntl* function was terminated prematurely by a signal.

C.2.25 lseek()

1003.1: (6.5.3.1) Specifies the function and its argument *offset* to be of type **off_t**.

SVID: Specifies the function and its argument *offset* to be of type **long**.

1003.1: Specifies the additional error condition

EINVAL The resulting file pointer would be illegal.

C.2.26 Terminal I/O

1003.1: Specifies the terminal control structure **termios**.

SVID: Specifies the terminal control structure **termio**.

1003.1: The Job Control Option is described. This includes changing the process group associated with the terminal, generating signals, SIGTTIN and SIGTTOU, for reads and writes from processes outside of the distinguished process group, generating a signal, SIGTSTP, upon receipt of a special character, SUSP, and a control flag, TOSTOP.

SVID: Volume 3 does not include the Job Control Option.

1003.1: (7.1.2.2) Specifies the types of the mode elements as **unsigned long**.

SVID: Specifies the types as **unsigned short**. Specifies a line discipline element *c_line*.

SVID: Specifies input mode flag IUCLC.

SVID: Specifies output mode flags OLCUC, ONLCR, OCRNL, ONOCR, ONLRET, OFILL, NLDLY, CRDLY, TABDLY, BSDLY, VTDLY, and FFDLY. Specifies delay values: NL0, NL1, CR0, CR1, CR2, CR3, TAB0, TAB1, TAB2, TAB3, BS0, BS1, VT0, VT1, FF0, and FF1.

1003.1: (7.1.2.5) Specifies the macros *cf_getospeed*(), *cf_setospeed*(), *cf_getispeed*(), and *cf_setispeed*() that get and set the input and output terminal speeds in a *termios* structure.

SVID: Specifies the local mode flag XCASE.

1003.1: (7.1.4) Specifies functions *tcgetattr*() and *tcsetattr*().

SVID: Specifies commands and structures for use with *ioctl*().

1003.1: (7.1.5) Specifies functions *tcsendbreak*(), *tcdrain*(), *tcflush*(), and *tcflow*(). The send-break function has the option of sending zero-valued bits for a specified value. The flow function has control over input.

SVID: Specifies commands and structures for use with *ioctl*().

D. Alternative Archive/Data Interchange Format

It has been proposed that the following section on the "Extended `tar` Format" be added to Chapter 10 as either an alternative to, or a replacement of, the "`cpio` Archive Format." Consult the cover letter for the ballot associated with this draft for an explanation of how to make your preferences known. Unless an explicit action is taken by the Balloting Group, this section will not appear in the approved Full Use Standard.

D.1 Extended `tar` Format

An extended `tar` archive tape or file contains a series of blocks. Each block is a fixed size block of TBLOCK bytes (see below). Although this format may be thought of as being stored on 9-track industry standard 1/2-inch magnetic tape, other types of transportable medium are not excluded. Each file archived is represented by a header block that describes the file, followed by zero or more blocks that give the contents of the file. At the end of the archive file are two blocks filled with binary zeros, interpreted as an end-of-archive indicator.

The blocks may be grouped for physical I/O operations. Each group of *n* blocks (where *n* is set by the application utility creating the archive file) may be written with a single *write*() operation. On magnetic tape, the result of this write is a single tape record. The last group of blocks is always at the full size, so blocks after the two zero blocks contain undefined data.

The header block is structured as follows. All lengths and offsets are in decimal.

Field Name	Byte Offset	Length (in bytes)
name	0	100
mode	100	8
uid	108	8
gid	116	8
size	124	12
mtime	136	12
chksum	148	8
typeflag	156	1
linkname	157	100
magic	257	6
version	263	2
uname	265	32
gname	297	32
devmajor	329	8
devminor	337	8
prefix	345	155

Symbolic constants used in the header block are defined in the header **<tar.h>** as follows:

```
#define TMAGIC  "ustar" /* ustar and a null */
#define TMAGLEN          6
#define TVERSION         "00"         /* 00 and no null */
#define TVERSLEN         2

/* Values used in typeflag field */
#define REGTYPE          '0'          /* Regular file  */
#define AREGTYPE         '\0'         /* Regular file  */
#define LNKTYPE          '1'          /* Link          */
#define SYMTYPE          '2'          /* Reserved      */
#define CHRTYPE          '3'          /* Char. special */
#define BLKTYPE          '4'          /* Block special */
#define DIRTYPE          '5'          /* Directory     */
#define FIFOTYPE         '6'          /* FIFO special  */
#define CONTTYPE         '7'          /* Reserved      */

/* Bits used in the mode field - values in octal */
#define TSUID      04000    /* Set UID on execution */
#define TSGID      02000    /* Set GID on execution */
#define TSVTX      01000    /* Reserved */
                            /* File permissions */
#define TUREAD     00400    /* read by owner */
#define TUWRITE    00200    /* write by owner */
#define TUEXEC     00100    /* execute/search by owner */
```

```
#define TGREAD    00040   /* read by group */
#define TGWRITE   00020   /* write by group */
#define TGEXEC    00010   /* execute/search by group */
#define TOREAD    00004   /* read by other */
#define TOWRITE   00002   /* write by other */
#define TOEXEC    00001   /* execute/search by other */
```

All characters are represented in the American Standard Code for Information Interchange, ASCII. For maximum portability between implementations, names should be picked from characters represented by the **portable filename character set** §2.3 as 8-bit characters with zero parity. If an extended character set beyond the portable character set is used, and the format-reading and format-creating utilities on the two distinct systems use the same extended character set, the file name shall be preserved. However, the format-reading utility shall never create file names on the local system that cannot be accessed via the functions calls described previously in this standard; see *open*() §5.3.1, *stat*() §5.6.2, *chdir*() §5.2.1, *fcntl*() §6.5.2, and *opendir*() §5.1.2. If a file name is found on the medium that would create an invalid file name, the implementation shall define if the data from the file in stored on the local file system and under what name it is stored. A format-reading utility may choose to ignore these files as long as it produces an error stating that the file is being ignored.

Each field within the header block is contiguous; that is, there is no padding used. Each character on the archive medium is stored contiguously.

The fields *magic*, *uname*, and *gname* are null-terminated character strings. The fields *name*, *linkname*, and *prefix* are null-terminated character strings except when all characters in the array contain non-null characters including the last character. All other fields are leading zero-filled octal numbers in ASCII. Each numeric field (of width w) contains $w-2$ digits, a space, and a null, except *size*, *mtime*, and *version*, that do not contain the trailing null.

The *name* and the *prefix* fields produce the pathname of the file. The hierarchical relationship of the file is retained by specifying the pathname as a path prefix, a slash character and filename as the suffix. If the *prefix* contains non-null characters, it is concatenated in front of the *name* without modification or addition of new characters to produce a new pathname. In this manner, pathnames of NAMSIZ plus PFXSIZ characters can be supported. If a pathname does not fit in the space provided, the format-creating utility shali notify the user of the error, and no attempt shall be made by the format-creating utility to store any part of the file, header or data, on the medium.

The *linkname* field, described below, does not use the *prefix* to produce a pathname. As such, a *linkname* is limited to NAMSIZ minus one characters. If the name does not fit in the space provided, the format-creating utility shall notify the user of the error, and the utility shall not attempt to store the link on the medium.

The *mode* field provides 9 bits specifying file permissions and 3 bits to specify the Set UID, Set GID, and TSVTX modes. Values for these bits are defined above. When special permissions are required to create a file with a given mode, and the user restoring files from the archive does not hold such permissions, the mode bit(s) requiring those special permissions are ignored. Modes not supported by the implementation restoring the files from the archive are ignored.

The *uid* and *gid* fields are the user and group ID of the file's owner and group, respectively.

The *size* field is the size of the file in bytes. If the *type* flag field is set to specify a file to be of type LNKTYPE, the *size* field shall be specified as a zero (0).

The *mtime* field is the modification time of the file at the time it was archived. It is the ASCII representation of the octal value of the modification time obtained from the *stat*() function.

The *chksum* field is the ASCII representation of the octal value of the simple sum of all bytes in the header block. Each 8-bit byte in the header is treated as an unsigned value. These values are added to an unsigned integer, initialized to zero, the precision of which shall be no less than 17 bits. When calculating the checksum, the *chksum* field is treated as if it were all blanks.

The *typeflag* field specifies the type of file archived. If a particular implementation does not recognize or permit the specified type, the file shall be extracted as if it were a regular file. As this action occurs, the format-reading utility shall issue a warning to the standard error output.

ASCII digit '0' represents a regular file.

For backward compatibility, a *typeflag* value of binary zero ('\0') should be recognized as meaning a regular file when extracting files from the archive. Archives written with this version of the archive file format shall create regular files with a *typeflag* value of ASCII '0'.

ASCII digit '1' represents a file linked to another file, of any type, previously archived. Such files are identified by each file having the same device and file serial number. The linked-to name is specified in the *linkname* field with a trailing null.

ASCII digit '2' is reserved.

ASCII digits '3' and '4' represent character special files and block special files respectively.

In this case the *devmajor* and *devminor* fields shall contain an encoding of the information found in the *st_rdev* field of the *stat* structure for the device file. Operating systems may map the device specifications to their own local specification, or may ignore the entry.

ASCII digit **'5'** specifies a directory or sub-directory. On systems where disk allocation is performed on a directory basis the *size* field shall contain the maximum number of bytes (which may be rounded to the nearest disk block allocation unit) that the directory may hold. A *size* field of zero indicates no such limiting. Systems that do not support limiting in this manner should ignore the *size* field.

ASCII digit **'6'** specifies a FIFO special file. Note that the archiving of a FIFO file archives the existence of this file and not its contents.

ASCII digit **'7'** is reserved.

ASCII letters **'A'** through **'Z'** are reserved for custom implementations. All other values are reserved for specification in future revisions of the standard.

The *magic* field is the specification that this archive was output in this archive format. If this field contains TMAGIC, then the *uname* and *gname* fields shall contain the ASCII representation of the owner and group of the file respectively. When the file is restored by a privileged, protection-preserving version of the utility, the password and group files shall be scanned for these names. If found, the user and group IDs contained within these files shall be used rather than the values contained within the *uid* and *gid* fields. c

The encoding of the header is designed to be portable across machines.

D.1.1 References

<grp.h> §9.2.1, **<pwd.h>** §9.2.2, **<sys/stat.h>** §5.6.1, *stat*() §5.6.2, **<unistd.h>** §2.10. B

E. Alternative *wait*() Functions

It has been proposed that the following section replace **Wait for Process Termination** §3.2.1. Consult the cover letter for the ballot associated with this draft for an explanation of how to make your preferences known. Unless an explicit action is taken by the Balloting Group, this section will not appear in the approved Full Use Standard.

E.1 Process Termination

...

E.1.1 Wait for Process Termination

Functions: *wait*(), *waitpid*()

E.1.1.1 Synopsis

```
int wait(stat_loc)
int *stat_loc;

int waitpid (stat_loc, pid, options)
int *stat_loc;
int pid;
int options;
```

E.1.1.2 Description

The header <sys/wait.h> defines the following arguments for the *waitpid*() function:

Constant	Description (*waitpid*() only)
WNOHANG	return immediately if no children to wait for
WUNTRACED	also return status for stopped children if the implementation supports the Job Control Option

If *stat_loc* is not (int *) 0, information called *status* shall be stored in the location pointed to by *stat_loc* as follows:

If the child process terminated due to an *_exit*() function, the low order 8 bits of *status* (corresponding to the octal value 0377) shall be zero, and the 8 bits corresponding to the octal value 0177400 shall contain the low order 8 bits of the argument that the child process passed to *_exit*() (see *_exit*() §3.2.2).

If the child process terminated due to a signal that was not caught, the low order 6 bits of *status* (corresponding to the octal value 077) shall contain the number of the signal that caused the termination, and the 8 bits corresponding to the octal value 0177400 shall be zero. In addition, if the bit that would be masked by the octal value 0200 is set, an abnormal termination with actions occurred (see *sigaction*() §3.3.4).

If the *wait*() function returned due to an implementation defined condition, the bit of *status* corresponding to the octal value 0100 shall be set. The value of the other bits of *status* are implementation defined and the child may not have terminated. If the child has terminated, a subsequent *wait*() function shall return its *status*.

If a parent process terminates without waiting for its child processes to terminate, its children shall be assigned a new parent process ID corresponding to an implementation defined system process. The *wait*() function shall only return successfully on the termination of a child process or due to an implementation defined change in status of a child process.

If the *waitpid*() variant is used, then the arguments *pid* and *options* are used to modify the behavior of the function.

If the *pid* argument specifies the child process for which *status* information is to be obtained, the process determined by *pid* is determined as follows:

- >0 — The *pid* specifies the process ID of a child process.
- 0 — The *pid* specifies any single child process whose process group ID is equal to that of the calling process.
- -1 — The *pid* specifies any single child process.
- <-1 — The *pid* specifies any single child process whose process group ID is equal to the absolute value of *pid*. The absolute value of *pid* shall not exceed {PID_MAX}.

The *options* argument contains two options that may be combined by forming their bitwise inclusive OR.

If the *options* bit indicated by WNOHANG is set, then *waitpid*() will not suspend the calling process if the process specified by *pid* has not terminated. If the implementation supports the Job Control Option, then the calling process specified by *pid* has not been stopped. In either case, a value of zero is returned by *waitpid*().

If the *options* bit indicated by WUNTRACED is set and if the implementation supports the Job Control Option, then *waitpid*() shall also return in *stat_loc* the wait status information when the process specified by *pid* is stopped due to a SIGTIN, SIGTOU, SIGTSTP, or SIGSTOP signal. In this case, the wait status information can also be interpreted in the following way:

> If the child process stopped, the 8 bits of *status* (corresponding to the octal value 0177400) shall contain the number of the signal that caused the process to stop and the low order 8 bits corresponding to the octal value 0377 shall be set equal to the octal value 0177.

If the implementation does not support the Job Control Option, then the WUNTRACED flag is ignored.

E.1.1.3 Returns

If the *wait*() function returns due to the receipt of a signal by the calling process, a value of −1 shall be returned to the calling process and *errno* shall be set to [EINTR]. If the *wait*() function returns due to a terminated child process, the process ID of the child shall be returned to the calling process. Otherwise, a value of −1 shall be returned, and *errno* shall be set to indicate the error.

If the *waitpid*() function returns due to the termination of a process specified by *pid*, the process ID of the terminated child shall be returned to the calling process.

If the implementation supports the Job Control Option and the *waitpid*() function is called with the WUNTRACED option, and the *waitpid*() function returns due to a process specified by *pid* having been stopped, the process ID of the stopped child shall be returned to the calling process.

If *waitpid*() is called and the WNOHANG option is used, then a value of zero shall be returned for one of two reasons:

1. The implemention supports the Job Control Option and the WUNTRACED option was used and the process specified by *pid* has not been stopped.

2. The process specified by *pid* has not been terminated.

Otherwise, the *waitpid*() function shall return a value of −1 and *errno* shall be set to indicate the error.

E.1.1.4 Errors

If any of the following conditions occur, the *wait*() and *waitpid*() functions shall return −1 and set *errno* to the corresponding value:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EINTR] The *wait*() function was terminated by a signal. The value pointed to by *stat_loc* may be undefined.

If any of the following conditions occur, the *waitpid*() function shall return −1 and set *errno* to the corresponding value:

[ECHILD] The process specified by *pid* is not a child process or does not exist.

[EINTR] The *waitpid*() function was terminated by a signal. The value pointed to by the *stat_loc* may be undefined.

[EINVAL] The *waitpid*() was called with an invalid *options* value.

E.1.1.5 References

exec §3.1.2, *_exit*() §3.2.2, *fork*() §3.1.1, *pause*() §3.4.2, *times*() §4.5.2, *sigaction*() §3.3.4.

Identifier Index

Topical Index

Acknowledgements

We wish to thank the following organizations for donating significant computer, printing, and editing resources to the production of this standard: /usr/group, Amdahl Corporation, Digital Equipment Corporation, and UniSoft Corporation.

Also we wish to thank the organizations employing the members of the Working Group and the Balloting Group for both covering the expenses related to attending and participating in meetings, and donating the time required both in and out of meetings for this effort.

Editor's Note: This list will be included in the final printed standard.

Company...
Company...
Company...
Company...