-- --

IEEE P1003.2 Draft 11.2 – September 1991

2.9 Dependencies on Other Standards

2.9.1 Features Inherited from POSIX.1

This subclause describes some of the features provided by POSIX.1 {8} that are assumed to be globally available by all systems conforming to POSIX.2. This subclause does not attempt to detail all of the POSIX.1 {8} features that are required by all of the utilities and functions defined in this standard; the utility and function descriptions point out additional functionality required to provide the corresponding specific features needed by each.

The following subclauses describe frequently used concepts. Utility and function description statements override these defaults when appropriate.

2.9.1.0.1 Features Inherited from POSIX.1 Rationale. *(This subclause is not a part of P1003.2)*

It has been pointed out that POSIX.2 assumes that a lot of POSIX.1 {8} functionality is present, but never states exactly how much. This is an attempt to clarify the assumptions.

This subclause only covers the ‘‘utilities and functions defined by this standard.’’ It does not mandate that the specific POSIX.1 {8} interfaces themselves be available to all application programs. A C language program compiled on a POSIX.2 system is not guaranteed that any of the POSIX.1 {8} functions are accessible. (For example, although UNIX system-based implementations of `ls` will use *stat*() to get file status, a POSIX.2 implementation of `ls` on a ‘LONG_NAME_OS-based’’ implementation might use the *get_file_attributes*() and the *get_file_time_stamps*() system calls.) POSIX.2 only requires equivalent functionality, not equal means of access. In any event, programs requiring the POSIX.1 {8} system interface should specify that they need POSIX.1 {8} conformance and not hope to achieve it by piggybacking on POSIX.2.

2.9.1.1 Process Attributes

The following process attributes, as described in POSIX.1 {8}, are assumed to be supported for all processes in POSIX.2:

controlling terminal	real group ID
current working directory	real user ID
effective group ID	root directory
effective user ID	saved set-group-ID
file descriptors	saved set-user-ID
file mode creation mask	session membership
process ID	supplementary group IDs
process group ID	

A conforming implementation may include additional process attributes.

2.9.1.1.1 Process Attributes Rationale. *(This subclause is not a part of P1003.2)*

The supplementary group IDs requirement is minimal. If {NGROUPS_MAX} is defined to be zero, they are not required. If {NGROUPS_MAX} is greater than zero, the supplementary group IDs are used as described in POSIX.1 {8} in various permission checking operations.

The saved-set-group-ID and saved-set-user-ID requirements are also minimal. If {_POSIX_SAVED_IDS} is defined, they are required; otherwise, they are not.

A controlling terminal is needed to control access to `/dev/tty`.

The file creation semantics of POSIX.2 require the effective group ID, effective user ID, and the file mode creation mask.

Pathname resolution and access permission checks require the current working directory, effective group ID, effective user ID, and root directory.

The `kill` utility requires the effective group ID, effective user ID, process ID, process group ID, real group ID, real user ID, saved set-group-ID, saved set-user-ID, and session membership attributes to perform the various signal addressing and permission checks.

The `id` utility is based on the effective group ID, effective user ID, real group ID, real user ID, and supplementary group IDs.

The following process attributes described in POSIX.1 {8} do not seem to be required by POSIX.2: parent process ID, pending signals, process signal mask, time left until an alarm clock signal, *tms_cstime*, *tms_cutime*, *tms_stime*, and *tms_utime*. There are probably other attributes mentioned in POSIX.1 {8} that are not listed here.

2.9.1.2 Concurrent Execution of Processes

The following functionality of the POSIX.1 {8} *fork*() function shall be available on all POSIX.2 conformant systems:

(1) Independent processes shall be capable of executing independently without either process terminating.

(2) A process shall be able to create a new process with all of the attributes referenced in 2.9.1.1, determined according to the semantics of a call to the POSIX.1 {8} *fork*() function followed by a call in the child process to one of the POSIX.1 {8} *exec* functions.

2.9.1.2.1 Concurrent Execution of Processes Rationale. *(This subclause is not a part of P1003.2)*

The historical functionality of *fork* () is required, which permits the concurrent execution of independent processes. A system with a single thread of process execution is not an appropriate base upon which to build a POSIX.2 system. (This requirement was not explicitly stated in the 1988 POSIX.1, but is included in the current POSIX.1 {8}.)

2.9.1.3 File Access Permissions

The file access control mechanism described by *file access permissions* in 2.2.2.55 applies to all files on a conforming POSIX.2 implementation.

2.9.1.3.1 File Access Permissions Rationale. *(This subclause is not a part of P1003.2)*

The entire concept of file protections and access control is assumed to be handled as in POSIX.1 {8}.

2.9.1.4 File Read, Write, and Creation

When a file is to be read or written, the file shall be opened with an access mode corresponding to the operation to be performed. If file access permissions deny access, the requested operation shall fail.

When a file that does not exist is created, the following POSIX.1 {8} features shall apply unless the utility or function description states otherwise:

(1) The file's user ID is set to the effective user ID of the calling process.

(2) The file's group ID is set to the effective group ID of the calling process or the group ID of the directory in which the file is being created.

(3) The file's permission bits are set to:

S_IROTH | S_IWOTH | S_IRGRP | S_IWGRP | S_IRUSR | S_IWUSR

(see POSIX.1 {8} 5.6.1.2) except that the bits specified by the process's file mode creation mask are cleared.

(4) The *st_atime*, *st_ctime*, and *st_mtime* fields of the file shall be updated as specified in *file times update* in 2.2.2.69.

(5) If the file is a directory, it shall be an empty directory; otherwise the file shall have length zero.

(6) Unless otherwise specified, the file created shall be a regular file.

When an attempt is made to create a file that already exists, the action shall depend on the file type:

(1) For directories and FIFO special files, the attempt shall fail and the utility shall either continue with its operation or exit immediately with a nonzero status, depending on the description of the utility.

(2) For regular files:

(a) The file's user ID, group ID, and permission bits shall not be changed.

(b) The file shall be truncated to zero length.

(c) The *st_ctime* and *st_mtime* fields shall be marked for update.

(3) For other file types, the effect is implementation defined.

When a file is to be appended, the file shall be opened in a manner equivalent to using the O_APPEND flag, without the O_TRUNC flag, in the POSIX.1 {8} *open*() call.

2.9.1.4.1 File Read, Write, and Creation Rationale. *(This subclause is not a part of P1003.2)*

Even though it might be possible for a process to change the mode of a file to match a requested operation and change the mode back to its original state after the operation is completed, utilities are not allowed to do this unless the utility description states otherwise. As an example, the `ed` utility `r` command fails if the file to be read does not exist (even though it could create the file and then read it) or the file permissions do not allow read access [even though it could use the POSIX.1 {8} *chmod*() function to make the file readable before attempting to open the file].

2.9.1.5 File Removal

When a directory that is the root directory or current working directory of any process is removed, the effect is implementation defined. If file access permissions deny access, the requested operation shall fail. Otherwise, when a file is removed:

(1) Its directory entry shall be removed from the file system.

(2) The link count of the file shall be decremented.

(3) If the file is an empty directory (see 2.2.2.43):

(a) If no process has the directory open, the space occupied by the directory shall be freed and the directory shall no longer be accessible.

(b) If one or more processes have the directory open, the directory contents shall be preserved until all references to the file have been closed.

(4) If the file is a directory that is not empty, the *st_ctime* field shall be marked for update.

(5) If the file is not a directory:

(a) If the link count becomes zero:

[1] If no process has the file open, the space occupied by the file shall be freed and the file shall no longer be accessible.

[2] If one or more processes have the file open, the file contents shall be preserved until all references to the file have been closed.

(b) If the link count is not reduced to zero, the *st_ctime* field shall be marked for update.

(6) The *st_ctime* and *st_mtime* fields of the containing directory shall be marked for update.

2.9.1.5.1 File Removal Rationale. *(This subclause is not a part of P1003.2)*

This is intended to be a summary of the POSIX.1 {8} *unlink*() and *rmdir*() requirements needed by POSIX.2.

2.9.1.6 File Time Values

All files have the three time values described by *file times update* in 2.2.2.69.

2.9.1.6.1 File Time Values Rationale. *(This subclause is not a part of P1003.2)*

All three time stamps specified by POSIX.1 {8} are needed for utilities like `find`, `ls`, `make`, `test`, and `touch` to work as expected.

2.9.1.7 File Contents

When a reference is made to the contents of a file, *pathname*, this means the equivalent of all of the data placed in the space pointed to by *buf* when performing the *read*() function calls in the following POSIX.1 {8} operations:

```
while (read (fildes, buf, nbytes) > 0)
        ;
```

If the file is indicated by a pathname *pathname*, the file descriptor shall be determined by the equivalent of the following POSIX.1 operation:

```
fildes = open (pathname, O_RDONLY);
```

The value of *nbytes* in the above sequence is unspecified; if the file is of a type where the data returned by *read*() would vary with different values, the value shall be one that results in the most data being returned.

If the *read*() function calls would return an error, it is unspecified whether the contents of the file are considered to include any data from offsets in the file beyond where the error would be returned.

2.9.1.7.1 File Contents Rationale. *(This subclause is not a part of P1003.2)*

This description is intended to convey the traditional behavior for all types of files. This matches the intuitive meaning for regular files, but the meaning is not always intuitive for other types of files. In particular, for FIFO s, pipes, and terminals it must be clear that the contents are not necessarily static at the time a file is opened, but they include the data returned by a sequence of reads until end-of-file is indicated. This is why the *open* () call is specified, with the O_NONBLOCK flag not set.

Some files, especially character special files, are sensitive to the size of a *read* () request. The contents of the file are those resulting from proper choice of this size.

2.9.1.8 Pathname Resolution

The pathname resolution algorithm described by *pathname resolution* in 2.2.2.104 shall be used by conforming POSIX. 2 implementations. See also *file hierarchy* in 2.2.2.58.

2.9.1.8.1 Pathname Resolution Rationale. *(This subclause is not a part of P1003.2)*

The whole concept of hierarchical file systems and pathname resolution is assumed to be handled as in POSIX .1 {8}.

2.9.1.9 Changing the Current Working Directory

When the current working directory (see 2.2.2.159) is to be changed, unless the utility or function description states otherwise, the operation shall succeed unless a call to the POSIX .1 {8} *chdir* () function would fail when invoked with the new working directory pathname as its argument.

2.9.1.9.1 Changing the Current Working Directory Rationale. *(This subclause is not a part of P1003.2)*

This subclause covers the access permissions and pathname structures involved with changing directories, such as with `cd` or (the UPE -extended) `mailx` utilities.

2.9.1.10 Establish the Locale

The functionality of the POSIX .1 {8} *setlocale* () function is assumed to be available on all POSIX. 2 conformant systems; i.e., utilities that require the capability of establishing an international operating environment shall be permitted to set the specified category of the international environment.

2.9.1.10.1 Establish the Locale Rationale. *(This subclause is not a part of P1003.2)*

The entire concept of locale categories such as the **LC_*** variables along with any implementation-defined categories is assumed to be handled as in POSIX.1 {8}.

2.9.1.11 Actions Equivalent to POSIX.1 Functions

Some utility descriptions specify that a utility performs actions equivalent to a POSIX.1 {8} function. Such specifications require only that the external effects be equivalent, not that any effect within the utility and visible only to the utility be equivalent.

2.9.1.11.1 Actions Equivalent to POSIX.1 Functions Rationale. *(This subclause is not a part of P1003.2)*

An objection was received to an earlier draft that said this approach of equivalent functions was unreasonable, as the reader (and the person writing a test suite) would be responsible for interpreting which portions of POSIX.1 {8} were included and which were not. For example, would such intermediate effects as the setting of *errno* be required if the related POSIX.1 {8} function called for that? The answer is no: this standard is only concerned with the end results of functions against the file system and the environment, and not any intermediate values or results visible only to the programmer using the POSIX.1 {8} function in a C (or other high-level language) program.

2.9.2 Concepts Derived from the C Standard

Some of the standard utilities perform complex data manipulation using their own procedure and arithmetic languages, as defined in their Extended Description or Operands subclauses. Unless otherwise noted, the arithmetic and semantic concepts (precision, type conversion, control flow, etc.) are equivalent to those defined in the C Standard {7}, as described in the following subclauses. Note that there is no requirement that the standard utilities be implemented in any particular programming language.

2.9.2.0.1 Concepts Derived from the C Standard Rationale. *(This subclause is not a part of P1003.2)*

This subclause was introduced to answer complaints that there was insufficient detail presented by such utilities as `awk` or `sh` about their procedural control statements and their methods of performing arithmetic functions. Earlier drafts, derived heavily from the original manual pages, contained statements such as ‘`for` loops similar to the C Standard {7},’’ which was good enough for a general understanding, but insufficient for a real implementation.

The C Standard {7} was selected as a model because most historical implementations of the standard utilities were written in C. Thus, it is more likely that they will act in a manner desired by POSIX.2 without modification.

Using the C Standard {7} is primarily a notational convenience, so the many ''little languages'' in POSIX.2 would not have to be rigorously described in every aspect. Its selection does not require that the standard utilities be written in Standard C; they could be written in common-usage C, Ada, Pascal, assembler language, or anything else.

The sizes of the various numeric values refer to C-language datatypes that are 1
allowed to be different sizes by the C Standard {7}. Thus, like a C-language appli- 1
cation, a shell application cannot rely on their exact size. However, it can rely on 1
their minimum sizes expressed in the C Standard {7}, such as {LONG_MAX} for a 1
long type. 1

2.9.2.1 Arithmetic Precision and Operations

Integer variables and constants, including the values of operands and option-arguments, used by the standard utilities shall be implemented as equivalent to the C Standard {7} *signed long* data type; floating point shall be implemented as equivalent to the C Standard {7} *double* type. Conversions between types shall be as described in the C Standard {7}. All variables shall be initialized to zero if they are not otherwise assigned by the application's input.

Arithmetic operators and functions shall be implemented as equivalent to those in the cited C Standard {7} section, as listed in Table 2-14.

The evaluation of arithmetic expressions shall be equivalent to that described in the C Standard {7} section 3.3 Expressions.

2.9.2.2 Mathematic Functions

Any mathematic functions with the same names as those in the C Standard {7}'s sections:

4.5 *Mathematics* <math.h>

4.10.2 *Pseudo-random sequence generation functions*

shall be implemented to return the results equivalent to those returned from a call to the corresponding C function described in the C Standard {7}.

Table 2-14 – C Standard Operators and Functions

Operation	C Standard {7} Equivalent Reference
( )	*3.3.1 Primary Expressions*
postfix ++ postfix −−	*3.3.2 Postfix Operators*
unary + unary − prefix ++ prefix −− ~ ! `sizeof()`	*3.3.3 Unary Operators*
* / %	*3.3.5 Multiplicative Operators*
+ −	*3.3.6 Additive Operators*
<< >>	*3.3.7 Bitwise Shift Operators*
<, <= >, >=	*3.3.8 Relational Operators*
== !=	*3.3.9 Equality Operators*
&	*3.3.10 Bitwise AND Operator*
ˆ	*3.3.11 Bitwise Exclusive OR Operator*
|	*3.3.12 Bitwise Inclusive OR Operator*
&&	*3.3.13 Logical AND Operator*
||	*3.3.14 Logical OR Operator*
expr ?expr :expr	*3.3.15 Conditional Operator*
=, *=, /=, %=, +=, −= <<=, >>=, &=, ˆ=, |=	*3.3.16 Assignment Operators*

`if ( )`
`if ( ) ... else`
`switch ( )`
	3.6.4 Selection Statements

`while ( )`
`do ... while ( )`
`for ( )`
	3.6.5 Iteration Statements

`goto`
`continue`
`break`
`return`
	3.6.6 Jump Statements

2.10 Utility Conventions

2.10.1 Utility Argument Syntax

This subclause describes the argument syntax of the standard utilities and introduces terminology used throughout the standard for describing the arguments processed by the utilities.

Within the standard, a special notation is used for describing the syntax of a utility's arguments. Unless otherwise noted, all utility descriptions use this notation, which is illustrated by this example (see 3.9.1):

> `utility_name [−a][−b][−c` *option_argument* `][−d | −e]`
> `[−f`*option_argument* `][`*operand* ... `]`

The notation used for the Synopsis subclauses imposes requirements on the implementors of the standard utilities and provides a simple reference for the reader of the standard.

(1) The utility in the example is named `utility_name`. It is followed by *options*, *option-arguments*, and *operands*. The arguments that consist of hyphens and single letters or digits, such as `−a`, are known as *options* (or, historically, *flags*). Certain options are followed by an *option-argument*, as shown with `[−c` *option_argument* `]`. The arguments following the last options and option-arguments are named *operands*.

(2) Option-arguments are sometimes shown separated from their options by `<blanks>`, sometimes directly adjacent. This reflects the situation that in some cases an option-argument is included within the same argument string as the option; in most cases it is the next argument. The Utility Syntax Guidelines in 2.10.2 require that the option be a separate argument from its option-argument, but there are some exceptions in this standard to ensure continued operation of historical applications:

   (a) If the Synopsis of a standard utility shows a `<space>` between an option and option-argument (as with `[−c` *option_argument* `]` in the example), a conforming application shall use separate arguments for that option and its option-argument.

   (b) If a `<space>` is not shown (as with `[−f`*option_argument* `]` in the example), a conforming application shall place an option and its option-argument directly adjacent in the same argument string, without intervening `<blank>`s.

   (c) Notwithstanding the requirements on conforming applications, a conforming implementation shall permit, but shall not require, an application to specify options and option-arguments as separate arguments whether or not a `<space>` is shown on the synopsis line.

   (d) A standard utility may also be implemented to operate correctly when the required separation into multiple arguments is violated by a nonconforming application.