-- --

IEEE P1003.2 Draft 11.2 − September 1991

```
#include <signal.h>

int system(const char *cmd)                                                     1
{
    int    stat;
    pid_t  pid;
    struct sigaction sa, savintr, savequit;
    sigset_t saveblock;

    if (cmd == NULL)
        return(1);
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigemptyset(&savintr.sa_mask);                                              1
    sigemptyset(&savequit.sa_mask);                                             1
    sigaction(SIGINT, &sa, &savintr);
    sigaction(SIGQUIT, &sa, &savequit);
    sigaddset(&sa.sa_mask, SIGCHLD);                                            1
    sigprocmask(SIG_BLOCK, &sa.sa_mask, &saveblock);
    if ((pid = fork()) == 0) {
        sigaction(SIGINT, &savintr, (struct sigaction *)0);
        sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
        sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
        execl("/bin/sh", "sh", "-c", cmd, (char *)0);
        _exit(127);
    }
    if (pid == -1) {
        stat = -1;  /* errno comes from fork() */
    } else {
        while (waitpid(pid, &stat, 0) == -1) {
            if (errno != EINTR) {
                stat = -1;
                break;
            }
        }
    }
    sigaction(SIGINT, &savintr, (struct sigaction *)0);
    sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
    sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
    return(stat);
}
```

Figure B-1 – Sample *system* () Implementation

Note that, while a particular implementation of *system* () (such as the one above) can assume a particular path for the shell, such a path is not necessarily valid on another system. The above example is not portable, and is not intended to be. There is no defined way for an application to find the specific path for the shell. However, *confstr* () can provide a value for **PATH** that is guaranteed to find the `sh` utility.

One reviewer suggested that an implementation of *system* () might want to use an environment variable such as **SHELL** to determine which command interpreter to use. The supposed implementation would use the default command interpreter if the one specified by the environment variable was not available. This would allow a user, when using an application that prompts for command lines to be processed using *system* (), to specify a different command interpreter. Such an implementation is discouraged. If the alternate command interpreter did not follow the command line syntax specified in POSIX.2, then changing **SHELL** would render *system* () nonconformant. This would affect applications that expected the specified behavior from *system* (), and since this standard does not mention that **SHELL** affects *system* (), the application would not know that it needed to unset **SHELL** .

B.3.2 C Binding for Pipe Communications with Programs

Functions: *popen* (), *pclose* ()

B.3.2.1 Synopsis

```
#include <stdio.h>

FILE *popen(const char *command , const char *mode );

int pclose(FILE *stream );
```

B.3.2.2 Description

The *popen* () function shall execute the command specified by the string *command* . It shall create a pipe between the calling program and the executed command, and return a pointer to a C Standard {7} stream that can be used to either read from or write to the pipe. The *pclose* () function shall close the stream, wait for the command to terminate, and return the termination status from the command language interpreter.

The environment of the executed command shall be as if a child process were created within the *popen* () call using the *fork* () function, and the child invoked the `sh` utility using the call:

```
execl(<shell path>, "sh", "-c", command , (char *)0);
```

where *<shell path>* is an unspecified pathname for the `sh` utility. However, 1
popen () shall ensure that any streams from previous *popen* () calls that remain 1
open in the parent process are closed in the new child process. 1

The *mode* argument to *popen* () is a string that specifies I/O mode:

(1) If *mode* is `"r"`, when the child process is started its file descriptor STDOUT_FILENO shall be the writable end of the pipe, and the file descriptor *fileno* (*stream*) in the calling process, where *stream* is the stream pointer returned by *popen* (), shall be the readable end of the pipe.

(2) If *mode* is `"w"`, when the child process is started its file descriptor STDIN_FILENO shall be the readable end of the pipe, and the file descriptor *fileno*(*stream*) in the calling process, where *stream* is the stream pointer returned by *popen*(), shall be the writable end of the pipe.

(3) If *mode* is any other value, the result is undefined.

A stream opened by *popen*() should be closed by *pclose*(). As stated above, *pclose*() shall return the termination status from the command language interpreter. However, if the application has called any of the following:

(1) *wait*(),

(2) *waitpid*() with a *pid* argument less than or equal to zero or equal to the process ID of the command line interpreter, or

(3) any other function not defined in POSIX.1 {8} or POSIX.2 that could do one of the above

and one of those calls caused the termination status to be unavailable to *pclose*(), then *pclose*() shall return −1 with *errno* set to [ECHILD] to report this situation. In any case, *pclose*() shall not return before the child process created by *popen*() has terminated.

If the command language interpreter cannot be executed, the child termination status returned by *pclose*() shall be as if the command language interpreter terminated using *exit*(127) or *_exit*(127). If it can be executed, the *exit*() value shall be as described for the `sh` utility.

The *pclose*() function shall not affect the termination status of any child of the calling process other than the one created by *popen*() for the associated stream.

If the argument *stream* to *pclose*() is not a pointer to a stream created by *popen*(), the result of *pclose*() is undefined.

After *popen*(), both the parent and the child process shall be capable of executing independently before either terminates. See 2.9.1.2.

B.3.2.3 Returns

The *popen*() function shall return a **NULL** pointer if the pipe or subprocess cannot be created. Otherwise, it shall return a stream pointer as described above.

Upon successful return, *pclose*() shall return the termination status of the command language interpreter. Otherwise, *pclose*() shall return −1 and set *errno* to indicate the error.

B.3.2.4 Errors

If any of the following conditions are detected, the *popen*() function shall return **NULL** and set *errno* to the corresponding value:

[EINVAL] The *mode* argument is invalid.

The *popen* () function may also set *errno* values as described by the POSIX.1 {8} *fork* () or *pipe* () functions.

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

[ECHILD] The status of the child process could not be obtained, as described above.

B.3.2.5 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

Because open files are shared, a mode `"r"` command can be used as an input filter and a mode `"w"` command as an output filter.

The behavior of *popen* () is specified for *mode* s of `"r"` and `"w"`. Other modes such as `"rb"` and `"wb"` might be supported by specific implementations, but these would not be portable features. Note that historical implementations of *popen* () only check to see if the first character of *mode* is `r`. Thus, a *mode* of `"robert the robot"` would be treated as *mode* `"r"`, and a *mode* of `"anything else"` would be treated as *mode* `"w"`.

If the application calls *waitpid* () with a *pid* argument greater than zero, and it still has a *popen* ()ed stream open, it must ensure that *pid* does not refer to the process started by *popen* ().

History of Decisions Made

There is a requirement that *pclose* () not return before the child process terminates. This is intended to disallow implementations that return [EINTR] if a signal is received while waiting. If *pclose* () returned before the child terminated, there would be no way for the application to discover which child used to be associated with the stream, and it could not do the cleanup itself.

If the stream pointed to by *stream* was not created by *popen* (), historical implementations of *pclose* () return −1 without setting *errno*. To avoid requiring *pclose* () to set *errno* in this case, this standard makes the behavior undefined. An application should not use *pclose* () to close any stream that wasn't created by *popen* ().

Wording was added in Draft 10 requiring that the parent and child processes be able to execute independently. This behavior has been the intent all along, and the specific words were taken from the current draft of the POSIX.1a revision to POSIX.1 {8}. Rationale about this wording appears in B.3.1.1 of POSIX.1a.

Some historical implementations either block or ignore the signals SIGINT, SIGQUIT, and SIGHUP while waiting for the child process to terminate. Since this behavior is not described in POSIX.2, such implementations are not conforming. Also, some historical implementations return [EINTR] if a signal is received, even though the child process has not terminated. Such implementations are also considered nonconforming.

Consider, for example, an application that uses

```
popen("command", "r")
```

to start *command*, which is part of the same application. The parent writes a prompt to its standard output (presumably the terminal) and then reads from the *popened* stream. The child reads the response from the user, does some transformation on the response (pathname expansion, perhaps) and writes the result to its standard output. The parent process reads the result from the pipe, does something with it, and prints another prompt. The cycle repeats. Assuming that both processes do appropriate buffer flushing, this would be expected to work.

Modified in Draft 6 to reflect the availability of the *waitpid*() function in POSIX.1 {8}. To conform to this standard, *pclose*() must use *waitpid*(), or some similar function, instead of *wait*().

Figure B-2 illustrates how the *pclose*() function might be implemented on a POSIX.1 {8} system.

```
int pclose(FILE *stream)                                          1
{
        int     stat;
        pid_t   pid;

        pid = <pid for process created for stream by popen()>
        (void) fclose(stream);

        while (waitpid(pid, &stat, 0) == -1) {
                if (errno != EINTR) {
                        stat = -1;
                        break;
                }
        }
        return(stat);
}
```

Figure B-2 – Sample *pclose*() Implementation

B.4 C Binding for Access Environment Variables

Function: *getenv* ()

The C language binding to the service described in 7.2 shall be the POSIX.1 {8} *getenv* () function.

B.5 C Binding for Regular Expression Matching

Functions: *regcomp* (), *regexec* (), *regfree* (), *regerror* ()

B.5.1 Synopsis

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

int regcomp(regex_t *preg, const char *pattern, int cflags);

int regexec(const regex_t *preg, const char *string,
        size_t nmatch, regmatch_t pmatch[], int eflags);

size_t regerror(int errcode, const regex_t *preg,
        char *errbuf, size_t errbuf_size);

void regfree(regex_t *preg);
```

B.5.2 Description

These functions shall interpret basic and extended regular expressions, as described in 2.8.

The header `<regex.h>` shall define the structure types *regex_t* and *regmatch_t*. The structure type *regex_t* shall include at least the member shown in Table B-6.

The structure type *regmatch_t* shall contain at least the members shown in
Table B-7. The type *regoff_t*, which shall be defined in `<regex.h>`, shall be a 1
signed arithmetic type that can hold the largest value that can be stored in either 1
an *off_t* or a *ssize_t*. 1

The *regcomp* () function shall compile the regular expression contained in the 1
string pointed to by the *pattern* argument and place the results in the structure 1
pointed to by *preg*. The *cflags* argument shall be the bitwise inclusive OR of zero or more of the flags shown in Table B-8, which shall be defined in the header `<regex.h>`.

The default regular expression type for *pattern* shall be a Basic Regular Expression. The application can specify Extended Regular Expressions using the REG_EXTENDED *cflags* flag.

If the function *regcomp* () succeeds, it shall return zero; otherwise it shall return nonzero, and the content of *preg* shall be undefined.

Table B-6 – Structure Type *regex_t*

Member Type	Member Name	Description
size_t	*re_nsub*	Number of parenthesized subexpressions.

Table B-7 – Structure Type *regmatch_t*

Member Type	Member Name	Description	
regoff_t	*rm_so*	Byte offset from start of *string* to start of substring.	1
regoff_t	*rm_eo*	Byte offset from start of *string* of the first character after the end of substring.	1 1

Table B-8 – *regcomp* () *cflags* Argument

flag	Description
REG_EXTENDED	Use Extended Regular Expressions.
REG_ICASE	Ignore case in match. See 2.8.2.
REG_NOSUB	Report only success/fail in *regexec* ().
REG_NEWLINE	Change the handling of `<newline>`, as described in the text.

Table B-9 – *regexec* () *eflags* Argument

flag	Description
REG_NOTBOL	The first character of the string pointed to by *string* is not the beginning of the line. Therefore, the circumflex character (^), when taken as a special character, shall not match the beginning of *string* .
REG_NOTEOL	The last character of the string pointed to by *string* is not the end of the line. Therefore, the dollar sign ($), when taken as a special character, shall not match the end of *string* .

If the REG_NOSUB flag was not set in *cflags* , then *regcomp* () shall set *re_nsub* to the number of parenthesized subexpressions [delimited by `\( \)` in basic regular expressions or `( )` in extended regular expressions] found in *pattern.*

The *regexec* () function shall compare the null-terminated string specified by *string* against the compiled regular expression *preg* initialized by a previous call to *regcomp* (). If it finds a match, *regexec* () shall return zero; otherwise it shall return nonzero indicating either no match or an error. The *eflags* argument shall be the bitwise inclusive OR of zero or more of the flags shown in Table B-9, which shall be defined in the header `<regex.h>`.

If *nmatch* is zero or REG_NOSUB was set in the *cflags* argument to *regcomp* (), then *regexec* () shall ignore the *pmatch* argument. Otherwise, the *pmatch* argument shall point to an array with at least *nmatch* elements, and *regexec* () shall fill in the elements of that array with offsets of the substrings of *string* that correspond to the parenthesized subexpressions of *pattern* : *pmatch* [*i*].*rm_so* shall be the byte offset of the beginning and *pmatch* [*i*].*rm_eo* shall be one greater than the byte offset of the end of substring *i*. (Subexpression *i* begins at the *i*th matched open parenthesis, counting from 1.) Offsets in *pmatch* [0] shall identify the substring that corresponds to the entire regular expression. Unused elements of *pmatch* up to *pmatch* [*nmatch* −1] shall be filled with −1. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself counts as a subexpression), then *regexec* () shall still do the match, but shall record only the first *nmatch* substrings.

When matching a basic or extended regular expression, any given parenthesized subexpression of *pattern* might participate in the match of several different substrings of *string* , or it might not match any substring even though the pattern as a whole did match. The following rules shall be used to determine which substrings to report in *pmatch* when matching regular expressions:

(1) If subexpression *i* in a regular expression is not contained within another subexpression, and it participated in the match several times, then the byte offsets in *pmatch* [*i*] shall delimit the last such match.

(2) If subexpression *i* is not contained within another subexpression, and it did not participate in an otherwise successful match, then the byte offsets in *pmatch* [*i*] shall be −1. A subexpression shall not participate in the match when:

   (a) * or `\{ \}` appears immediately after the subexpression in a basic regular expression, or *, ?, or `{ }` appears immediately after the subexpression in an extended regular expression, and the subexpression did not match (matched zero times), or

   (b) **|** is used in an extended regular expression to select this subexpression or another, and the other subexpression matched.

(3) If subexpression *i* is contained within another subexpression *j*, and *i* is not contained within any other subexpression that is contained within *j*, and a match of subexpression *j* is reported in *pmatch* [*j*], then the match or nonmatch of subexpression *i* reported in *pmatch* [*i*] shall be as described in (1) and (2) above, but within the substring reported in *pmatch* [*j*] rather than the whole string.

(4) If subexpression *i* is contained in subexpression *j*, and the byte offsets in *pmatch* [*j*] are −1, then the byte offsets in *pmatch* [*i*] also shall be −1.

(5) If subexpression *i* matched a zero-length string, then both byte offsets in *pmatch* [*i*] shall be the byte offset of the character or null terminator immediately following the zero-length string.

If, when *regexec* () is called, the locale is different than when the regular expression was compiled, the result is undefined.

If REG_NEWLINE is not set in *cflags*, then a <newline> character in *pattern* or *string* shall be treated as an ordinary character. If REG_NEWLINE is set, then <newline> shall be treated as an ordinary character except as follows:

(1) A <newline> in *string* shall not be matched by a period outside of a bracket expression (see 2.8.3.1.3) or by any form of a nonmatching list (see 2.8.3.2).

(2) A circumflex (^) in *pattern*, when used to specify expression anchoring (see 2.8.4.4 and 2.8.4.6), shall match the zero-length string immediately after a <newline> in *string*, regardless of the setting of REG_NOTBOL.

(3) A dollar-sign ($) in *pattern*, when used to specify expression anchoring, shall match the zero-length string immediately before a <newline> in *string*, regardless of the setting of REG_NOTEOL.

The *regfree*() function shall free any memory allocated by *regcomp*() associated with *preg*.

The *regerror*() function provides a mapping from error codes returned by *regcomp*() and *regexec*() to unspecified printable strings. It shall generate a string corresponding to the value of the *errcode* argument, which shall be the last nonzero value returned by *regcomp*() or *regexec*() with the given value of *preg*. If
errcode is not such a value, the content of the generated string is unspecified. If 1
preg is (*regexec_t*)0, but *errcode* is a value returned by a previous call to *regexec*() 1
or *regcomp*(), then *regerror*() still shall generate an error string corresponding to 1
the value of *errcode*, but it might not be as detailed under some implementations. 1

If the *errbuf_size* argument is not zero, *regerror*() shall place the generated string into the *errbuf_size*-byte buffer pointed to by *errbuf*. If the string (including the terminating null) cannot fit in the buffer, *regerror*() shall truncate the string and null-terminate the result.

If *errbuf_size* is zero, *regerror*() shall ignore the *errbuf* argument, but shall return the integer value described below.

If the *preg* argument to *regexec*() or *regfree*() is not a compiled regular expression returned by *regcomp*(), the result is undefined. A *preg* shall no longer be treated as a compiled regular expression after it is given to *regfree*().

B.5.3 Returns

On successful completion, the *regcomp*() function shall return zero. On successful completion, the *regexec*() function shall return zero to indicate that *string* matched *pattern*, or REG_NOMATCH (which shall be defined in <regex.h>) to indicate no match.

The *regerror*() function shall return the size of the buffer needed to hold the entire generated string, including the null termination. If the return value is greater than *errbuf_size*, the string returned in the buffer pointed to by *errbuf* has been truncated.

Table B-10 – *regcomp* (), *regexec* () Return Values

Error Code	Description
REG_NOMATCH	*regexec* () failed to match
REG_BADPAT	Invalid regular expression
REG_ECOLLATE	Invalid collating element referenced
REG_ECTYPE	Invalid character class type referenced
REG_EESCAPE	Trailing \ in pattern
REG_ESUBREG	Number in \ *digit* invalid or in error
REG_EBRACK	[ ] imbalance
REG_EPAREN	\(\) or ( ) imbalance
REG_EBRACE	\{ \} imbalance
REG_BADBR	Content of \{ \} invalid: Not a number, number too large, more than two numbers, first larger than second
REG_ERANGE	Invalid endpoint in range expression
REG_ESPACE	Out of memory
REG_BADRPT	?, *, or + not preceded by valid regular expression

B.5.4 Errors

If *regcomp* () or *regexec* () fails, it shall return a nonzero value indicating the type of failure. Table B-10 contains the names of macros for error codes that may be returned. If a code is returned, the interpretation shall be as given in the table. The implementation shall define the macros in Table B-10 in <regex.h>, and may define additional macros beginning with 'REG_'' for other error codes.

If *regcomp* () detects an illegal regular expression, it may return REG_BADPAT, or it may return one of the error codes that more precisely describes the error.

B.5.5 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

An example of using the functions is shown in Figure B-3

The following demonstrates how the REG_NOTBOL flag could be used with *regexec* () to find all substrings in a line that match a pattern supplied by a user. (For simplicity of the example, very little error checking is done.)

```
(void) regcomp (&re, pattern, 0);
/* this call to regexec() finds the first match on the line */
error = regexec (&re, &buffer[0], 1, &pm, 0);
while (error == 0) {    /* while matches found */
        <substring found between pm.rm_sp and pm.rm_ep>
        /* This call to regexec() finds the next match */
        error = regexec (&re, pm.rm_ep, 1, &pm, REG_NOTBOL);
}
```

```
#include <regex.h>

/*
 * Match string against the extended regular expression in
 * pattern, treating errors as no match.
 *
 * Return 1 for match, 0 for no match.
 */

int
match(const char *string, const char *pattern)
{
        int     status;
        regex_t re;

        if (regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB) != 0) {
                return(0);      /* report error */
        }
        status = regexec(&re, string, (size_t) 0, NULL, 0);
        regfree(&re);
        if (status != 0) {
                return(0);      /* report error */
        }
        return status == 0;
}
```

Figure B-3 – Example Regular Expression Matching

An application could use `regerror(code,preg,NULL,(size_t)0)` to find out how big a buffer is needed for the generated string, *malloc* () a buffer to hold the string, and then call *regerror* () again to get the string. Alternately, it could allocate a fixed, static buffer that is big enough to hold most strings (perhaps 128 bytes), and then *malloc* () a larger buffer if it finds that this is too small.

The *regmatch* () function must fill in all *nmatch* elements of *pmatch* , where *nmatch* and *pmatch* are supplied by the application, even if some elements of *pmatch* do not correspond to subexpressions in *pattern* . The application writer should note that there is probably no reason for using a value of *nmatch* that is larger than *preg->re_nsub* .

History of Decisions Made

The REG_ICASE flag supports the operations taken by the `grep -i` option and the historical implementations of `ex` and `vi`. Including this flag will make it easier for application code to be written that does the same thing as these utilities.

The substrings reported in *pmatch* [] are defined using offsets from the start of the string rather than pointers. Since this is a new interface, there should be no impact on historical implementations or applications, and offsets should be just as easy to use as pointers. The change to offsets was made to facilitate future extensions in which the string to be searched is presented to *regexec*() in blocks, allowing a string to be searched that is not all in memory at once.

A new type *regoff_t* is used for the elements of *pmatch*[] to ensure that the application can represent either the largest possible array in memory (important for a POSIX.2-conforming application) or the largest possible file (important for an application using the extension where a file is searched in chunks).

The working group has rejected, at least for now, the inclusion of a *regsub*() function that would be used to do substitutions for a matched regular expression. While such a routine would be useful to some applications, its utility would be much more limited than the matching function described here. Both regular expression parsing and substitution are possible to implement without support other than that required by the C Standard {7}, but matching is much more complex than substituting. The only ‘‘difficult’’ part of substitution, given the information supplied by *regexec*(), is finding the next character in a string when there can be multibyte characters. That is a much wider issue, and one that needs a more general solution.

The *errno* variable has not been used for error returns to avoid cluttering up the *errno* namespace for this feature.

In Draft 9, the interface was modified so that the matched substrings *rm_sp* and *rm_ep* are in a separate *regmatch_t* structure instead of in *regex_t*. This allows a single compiled regular expression to be used simultaneously in several contexts; in *main*() and a signal handler, perhaps, or in multiple threads of lightweight processes. (The *preg* argument to *regexec*() is declared with type `const`, so the implementation is not permitted to use the structure to store intermediate results.) It also allows an application to request an arbitrary number of substrings from a regular expression. (Previous versions reported only ten substrings.) The number of subexpressions in the regular expression is reported in *re_nsub* in *preg*. With this change to *regexec*(), consideration was given to dropping the REG_NOSUB flag, since the user can now specify this with a zero *nmatch* argument to *regexec*(). However, keeping REG_NOSUB allows an implementation to use a different (perhaps more efficient) algorithm if it knows in *regcomp*() that no subexpressions need be reported. The implementation is only required to fill in *pmatch* if *nmatch* is not zero and if REG_NOSUB is not specified. Note that the *size_t* type, as defined in the C Standard {7}, is unsigned, so the description of *regexec*() does not need to address negative values of *nmatch*.

The rules for reporting substrings of extended regular expressions are consistent with those used by Henry Spencer's ‘‘almost public domain’’ version of *regexec*().

The REG_NOTBOL and REG_NOTEOL flags were added to *regexec*() in Draft 9. REG_NOTBOL was added to allow an application to do repeated searches for the same pattern in a line. If the pattern contains a circumflex character that should match the beginning of a line, then the pattern should only match when matched against the beginning of the line. Without the REG_NOTBOL flag, the application could rewrite the expression for subsequent matches, but in the general case this would require parsing the expression. The need for REG_NOTEOL is not as clear;

it was added for symmetry.

The addition of the *regerror* () function addresses the historical need for portable application programs to have access to error information more than ‘‘Function failed to compile/match your regular expression for unknown reasons.’’

This interface provides for two different methods of dealing with error conditions. The specific error codes (REG_EBRACE, for example), defined in `<regex.h>`, allow an application to recover from an error if it is so able. Many applications, especially those that use patterns supplied by a user, will not try to deal with specific error cases, but will just use *regerror* () to obtain a human-readable error message to present to the user.

The *regerror* () function uses a scheme similar to *confstr* () to deal with the problem of allocating memory to hold the generated string. The scheme used by *strerror* () in the C Standard {7} was considered unacceptable since it creates difficulties for multithreaded applications. (POSIX.4a, a standard for threads, started balloting in January 1991.) A different scheme used by *regerror* () in one draft of this standard was eliminated to improve internal consistency, and because the current interface produced greater consensus than the other.

The *preg* argument is provided to *regerror* () to allow an implementation to generate a more descriptive message than would be possible with *errcode* alone. An implementation might, for example, save the character offset of the offending character of the pattern in a field of *preg*, and then include that in the generated message string. The implementation may also ignore *preg*.

A REG_FILENAME flag was considered, but omitted. This flag caused *regexec* () to match patterns as described in 3.13 instead of regular expressions. This service is now provided by the *fnmatch* () function [see B.6].

B.6 C Binding for Match Filename or Pathname

Function: *fnmatch* ()

B.6.1 Synopsis

```
#include <fnmatch.h>
int fnmatch(const char *pattern, const char *string, int flags);
```

B.6.2 Description

The *fnmatch* () function shall match patterns as described in 3.13.1 and 3.13.2. It checks the string specified by the *string* argument to see if it matches the pattern specified by the *pattern* argument.

The *flags* argument modifies the interpretation of *pattern* and *string*. It is the bitwise inclusive OR of zero or more of the flags shown in Table B-11, which are defined in the header `<fnmatch.h>`. If the FNM_PATHNAME flag is set in *flags*, then a slash character in *string* shall be explicitly matched by a slash in *pattern*; it shall not be matched by either the asterisk or question-mark special characters, nor by a bracket expression. If the FNM_PATHNAME flag is not set, the slash character shall be treated as an ordinary character.

Table B-11 – *fnmatch* () *flags* Argument

flags	Description
FNM_NOESCAPE	Disable backslash escaping
FNM_PATHNAME	Slash in *string* only matches slash in *pattern*
FNM_PERIOD	Leading period in *string* must be exactly matched by period in *pattern*

If FNM_NOESCAPE is not set in *flags*, a backslash character (\) in *pattern* followed by any other character shall match that second character in *string*. In particular, `'\\'` shall match a backslash in *string*. If FNM_NOESCAPE is set, a backslash character shall be treated as an ordinary character.

If FNM_PERIOD is set in *flags*, then a leading period in *string* shall match a period in *pattern* as described by rule (2) in 3.13.2, where the location of ‘‘leading’’ is indicated by the value of FNM_PATHNAME:

— If FNM_PATHNAME is set, a period is ‘‘leading’’ if it is the first character in *string* or if it immediately follows a slash.

— If FNM_PATHNAME is not set, a period is ‘‘leading’’ only if it is the first character of *string*.

If FNM_PERIOD is not set, then no special restrictions shall be placed on matching a period.

B.6.3 Returns

If *string* matches the pattern specified by *pattern*, then *fnmatch*() shall return zero. If there is no match, *fnmatch*() shall return FNM_NOMATCH, which shall be defined in the header `<fnmatch.h>`. If an error occurs, *fnmatch*() shall return another nonzero value.

B.6.4 Errors

This standard does not specify any error conditions that are required to be detected by the *fnmatch*() function. Some errors may be detected under unspecified conditions.

B.6.5 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The *fnmatch*() function has two major uses. It could be used by an application or utility that needs to read a directory and apply a pattern against each entry. The `find` utility is an example of this. It can also be used by the `pax` utility to process its *pattern* operands, or by applications that need to match strings in a similar manner.

History of Decisions Made

This function replaces the REG_FILENAME flag of *regcomp*() in early drafts. It provides virtually the same functionality as the *regcomp*() and *regexec*() functions using the REG_FILENAME and REG_FSLASH flags [the REG_FSLASH flag was proposed for *regcomp*(), and would have had the opposite effect from FMN_PATHNAME], but with a simpler interface and less overhead.

The name *fnmatch*() is intended to imply *filename* match, rather than *pathname* match. The default action of this function is to match filenames, rather than pathnames, since it gives no special significance to the slash character. With the FNM_PATHNAME flag, *fnmatch*() does match pathnames, but without tilde expansion, parameter expansion, or special treatment for period at the beginning of a filename.

B.7 C Binding for Command Option Parsing

Function: *getopt* ()

B.7.1 Synopsis

```
#include <unistd.h>

int getopt(int argc, char * const argv[], const char *optstring);

extern char *optarg;
extern int optind, opterr, optopt;
```

B.7.2 Description

The *getopt* () function is a command-line parser that can be used by applications that follow Utility Syntax Guidelines 3, 4, 5, 6, 7, 9, and 10 in 2.10.2. The remaining guidelines are not addressed by *getopt* () and are the responsibility of the application.

The parameters *argc* and *argv* are the argument count and argument array as passed to *main* (). The argument *optstring* is a string of recognized option characters; if a character is followed by a colon, the option takes an argument. All option characters allowed by Utility Syntax Guideline 3 are allowed in *optstring* . The implementation may accept other characters as an extension.

The variable *optind* is the index of the next element of the *argv* [] vector to be processed. It is initialized to 1 by the system, and *getopt* () updates it when it finishes with each element of *argv* []. When an element of *argv* [] contains multiple option characters, it is unspecified how *getopt* () determines which options have already been processed.

The *getopt* () function shall return the next option character from *argv* that matches a character in *optstring* , if there is one that matches. If the option takes an argument, *getopt* () shall set the variable *optarg* to point to the option-argument as follows:

(1) If the option was the last character in the string pointed to by an element of *argv* , then *optarg* contains the next element of *argv* , and *optind* shall be incremented by 2. If the resulting value of *optind* is not less than *argc* , this indicates a missing option argument, and *getopt* () shall return an error indication.

(2) Otherwise, *optarg* points to the string following the option character in that element of *argv* , and *optind* shall be incremented by 1.

If, when *getopt* () is called, *argv* [*optind*] is **NULL** , **argv* [*optind*] is not the character –, or *argv* [*optind*] points to the string `"-"`, *getopt* () shall return −1 without changing *optind* . If *argv* [*optind*] points to the string `"--"`, *getopt* () shall return −1 after incrementing *optind* .

If *getopt* () encounters an option character that is not contained in *optstring* , it shall return the question-mark (`?`) character. If it detects a missing option argument, it shall return the colon character (`:`) if the first character of *optstring* was a colon, or a question-mark character otherwise. In either case, *getopt* () shall set the variable *optopt* to the option character that caused the error. If the application has not set the variable *opterr* to zero and the first character of *optstring* is not a colon, *getopt* () shall also print a diagnostic message to standard error using 1
the formatting rules specified for the `getopts` utility (see 4.27.6.2). 1

B.7.3 Returns

The *getopt* () function shall return the next option character specified on the command line. The value −1 shall be returned when all command line options have been parsed.

B.7.4 Errors

If an invalid option is encountered, *getopt* () shall return a question-mark character. If an option with a missing option argument is encountered, *getopt* () shall return either a question-mark or a colon, as described previously.

B.7.5 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The *getopt* () function is only required to support option characters included in Guideline 3. Many historical implementations of *getopt* () support other characters as options. This is an allowed extension, but applications that use extensions are not maximally portable. Note that support for multibyte option characters is only possible when such characters can be represented as type *int* .

The code fragment in Figure B-4 shows how one might process the arguments for a utility that can take the mutually exclusive options `a` and `b` and the options `f` and `o`, both of which require arguments.

The code in Figure B-4 accepts any of the following as equivalent:

```
cmd -ao arg path path
cmd -a -o arg path path
cmd -o arg -a path path
cmd -a -o arg -- path path
cmd -a -oarg path path
cmd -aoarg path path
```

History of Decisions Made

Support for the *optopt* variable was added in Draft 9. This documents historical practice, and allows the application to obtain the identity of the invalid option.

```
#include  <unistd.h>
int main (int argc, char *argv[ ])
{
        int c, bflg, aflg, errflg = 0;
        char *ifile, *ofile;
        extern char *optarg;
        extern int optind, optopt;
        . . .
        while ((c = getopt(argc, argv, ":abf:o:")) != -1) {
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg = 1;
                        else
                                aflg = 1;
                        break;
                case 'b':
                        if (aflg)
                                errflg = 1;
                        else
                                bflg = 1;
                                bproc( );
                        break;
                case 'f':
                        ifile = optarg;
                        break;
                case 'o':
                        ofile = optarg;
                        break;
                case ':':       /* -f or -o without option-arg */
                        fprintf (stderr,
                                "Option -%c requires an option-argument\n",
                                optopt);
                        errflg = 1;
                        break;
                case '?':
                        fprintf (stderr,
                                "Unrecognized option: -%c\n", optopt);
                        errflg = 1;
                        break;
                }
        }
        if (errflg) {
                fprintf(stderr, "usage: . . . ");
                exit(2);
        }
        for ( ; optind < argc; optind++) {
                if (access(argv[optind], R_OK)) {
        . . .
}
```

Figure B-4 – Argument Processing with *getopt* ()

The description was extensively rewritten in Draft 9 to be more explicit about how *optarg* and *optind* are set, and to recognize that this routine deals with a vector of string pointers, not directly with a shell command line.

The description was modified in Draft 9 to make it clear that *getopt* (), like the `getopts` utility, shall deal with option-arguments whether separated from the option by `<blank>`s or not. Note that the requirements on *getopt* () and `getopts` are more stringent than the Utility Syntax Guidelines.

The *getopt* () function has been changed to return `−1`, rather than `EOF`, so that *<stdio.h>* is not required.

The special significance of a colon as the first character of *optstring* was added in 1
Draft 11 to make *getopt* () consistent with the `getopts` utility. It allows an appli- 1
cation to make a distinction between a missing argument and an incorrect option 1
letter without having to examine the option letter. It is true that a missing argu- 1
ment can only be detected in one case, but that is a case that has to be considered. 1

B.8 C Binding for Generate Pathnames Matching a Pattern

Functions: *glob* (), *globfree* ()

B.8.1 Synopsis

```
#include <glob.h>

int glob(const char *pattern , int flags ,
        int (*errfunc )(const char *epath , int eerrno ), glob_t *pglob);

void globfree(glob_t *pglob);
```

B.8.2 Description

The *glob* () function is a pathname generator that implements the rules defined in 3.13, with optional support for rule (3) in 3.13.3.

The header `<glob.h>` defines the structure type *glob_t* , which includes at least the members shown in Table B-12.

The argument *pattern* is a pointer to a pathname pattern to be expanded. The *glob* () function shall match all accessible pathnames against this pattern and develop a list of all pathnames that match. In order to have access to a pathname, *glob* () requires search permission on every component of a path except the last and read permission on each directory of any filename component of *pattern* that contains any of the special characters `*`, `?` or `[`. The *glob* () function stores the number of matched pathnames into *pglob->gl_pathc* and a pointer to a list of pointers to pathnames into *pglob->gl_pathv* . The pathnames are in sort order as defined by 2.2.2.30. The first pointer after the last pathname shall be **NULL** . If the pattern does not match any pathnames, the returned number of matched paths is set to zero.

Table B-12 – Structure Type *glob_t*

Member Type	Member Name	Description
size_t	*gl_pathc*	Count of paths matched by *pattern*.
*char ***	*gl_pathv*	Pointer to a list of matched pathnames.
size_t	*gl_offs*	Slots to reserve at the beginning of *gl_pathv*.

It is the caller's responsibility to create the structure pointed to by *pglob*. The *glob*() function shall allocate other space as needed, including the memory pointed to by *gl_pathv*. The *globfree*() function shall free any space associated with *pglob* from a previous call to *glob*().

The argument *flags* is used to control the behavior of *glob*(). The value of *flags* is the bitwise inclusive OR of any of the constants shown in Table B-13, which are defined in `<glob.h>`.

Table B-13 – *glob*() *flags* Argument

Name	Description
GLOB_APPEND	Append pathnames generated to the ones from a previous call to *glob*().
GLOB_DOOFFS	Make use of *pglob->gl_offs*. If this flag is set, *pglob->gl_offs* is used to specify how many **NULL** pointers to add to the beginning of *pglob->gl_pathv*. In other words, *pglob->gl_pathv* shall point to *pglob->gl_offs* **NULL** pointers, followed by *pglob->gl_pathc* pathname pointers, followed by a **NULL** pointer.
GLOB_ERR	Causes *glob*() to return when it encounters a directory that it cannot open or read. Ordinarily, *glob*() continues to find matches.
GLOB_MARK	Each pathname that is a directory that matches *pattern* has a slash appended.
GLOB_NOCHECK	Support rule (3) in 3.13.3. If *pattern* does not match any pathname, then *glob*() shall return a list consisting of only *pattern*, and the number of matched pathnames is 1.
GLOB_NOESCAPE	Disable backslash escaping.
GLOB_NOSORT	Ordinarily, *glob*() sorts the matching pathnames according to the definition of *collation sequence* in 2.2.2.30. When this flag is used the order of pathnames returned is unspecified.

The GLOB_APPEND flag can be used to append a new set of words to those generated by a previous call to *glob*(). The following rules apply when two or more calls to *glob*() are made with the same value of *pglob* and without intervening calls to *globfree*():

(1) The first such call shall not set GLOB_APPEND. All subsequent calls shall set it.