7.19 Input/output <stdio.h>

7.19.1 Introduction

1 The header **<stdio.h>** declares three types, several macros, and many functions for performing input and output.

2 The types declared are **size_t** (described in 7.17);

FILE

which is an object type capable of recording all the information needed to control a stream, including its file position indicator, a pointer to its associated buffer (if any), an *error indicator* that records whether a read/write error has occurred, and an *end-of-file indicator* that records whether the end of the file has been reached; and

fpos_t

which is an object type other than an array type capable of recording all the information needed to specify uniquely every position within a file.

3 The macros are **NULL** (described in 7.17);

_IOFBF
_IOLBF
_IONBF

which expand to integer constant expressions with distinct values, suitable for use as the third argument to the **setvbuf** function;

BUFSIZ

which expands to an integer constant expression that is the size of the buffer used by the **setbuf** function;

EOF

which expands to an integer constant expression, with type **int** and a negative value, that is returned by several functions to indicate *end-of-file*, that is, no more input from a stream;

FOPEN_MAX

which expands to an integer constant expression that is the minimum number of files that the implementation guarantees can be open simultaneously;

FILENAME_MAX

which expands to an integer constant expression that is the size needed for an array of **char** large enough to hold the longest file name string that the implementation

guarantees can be opened;[222)]

L_tmpnam

which expands to an integer constant expression that is the size needed for an array of **char** large enough to hold a temporary file name string generated by the **tmpnam** function;

SEEK_CUR
SEEK_END
SEEK_SET

which expand to integer constant expressions with distinct values, suitable for use as the third argument to the **fseek** function;

TMP_MAX

which expands to an integer constant expression that is the maximum number of unique file names that can be generated by the **tmpnam** function;

stderr
stdin
stdout

which are expressions of type ''pointer to **FILE**'' that point to the **FILE** objects associated, respectively, with the standard error, input, and output streams.

4 The header **<wchar.h>** declares a number of functions useful for wide character input and output. The wide character input/output functions described in that subclause provide operations analogous to most of those described here, except that the fundamental units internal to the program are wide characters. The external representation (in the file) is a sequence of ''generalized'' multibyte characters, as described further in 7.19.3.

5 The input/output functions are given the following collective terms:

— The *wide character input functions* — those functions described in 7.24 that perform input into wide characters and wide strings: **fgetwc**, **fgetws**, **getwc**, **getwchar**, **fwscanf**, **wscanf**, **vfwscanf**, and **vwscanf**.

— The *wide character output functions* — those functions described in 7.24 that perform output from wide characters and wide strings: **fputwc**, **fputws**, **putwc**, **putwchar**, **fwprintf**, **wprintf**, **vfwprintf**, and **vwprintf**.

---

222) If the implementation imposes no practical limit on the length of file name strings, the value of **FILENAME_MAX** should instead be the recommended size of an array intended to hold a file name string. Of course, file name string contents are subject to other system-specific constraints; therefore *all* possible strings of length **FILENAME_MAX** cannot be expected to be opened successfully.

— The *wide character input/output functions* — the union of the **ungetwc** function, the wide character input functions, and the wide character output functions.

— The *byte input/output functions* — those functions described in this subclause that perform input/output: **fgetc**, **fgets**, **fprintf**, **fputc**, **fputs**, **fread**, **fscanf**, **fwrite**, **getc**, **getchar**, **gets**, **printf**, **putc**, **putchar**, **puts**, **scanf**, **ungetc**, **vfprintf**, **vfscanf**, **vprintf**, and **vscanf**.

Forward references: files (7.19.3), the **fseek** function (7.19.9.2), streams (7.19.2), the **tmpnam** function (7.19.4.4), **<wchar.h>** (7.24).

7.19.2 Streams

1 Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data *streams*, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported, for *text streams* and for *binary streams*.[223)]

2 A text stream is an ordered sequence of characters composed into *lines*, each line consisting of zero or more characters plus a terminating new-line character. Whether the last line requires a terminating new-line character is implementation-defined. Characters may have to be added, altered, or deleted on input and output to conform to differing conventions for representing text in the host environment. Thus, there need not be a one-to-one correspondence between the characters in a stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if: the data consist only of printing characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is implementation-defined.

3 A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream, under the same implementation. Such a stream may, however, have an implementation-defined number of null characters appended to the end of the stream.

4 Each stream has an *orientation*. After a stream is associated with an external file, but before any operations are performed on it, the stream is without orientation. Once a wide character input/output function has been applied to a stream without orientation, the

223) An implementation need not distinguish between text streams and binary streams. In such an implementation, there need be no new-line characters in a text stream nor any limit to the length of a line.

stream becomes a *wide-oriented stream*. Similarly, once a byte input/output function has been applied to a stream without orientation, the stream becomes a *byte-oriented stream*. Only a call to the **freopen** function or the **fwide** function can otherwise alter the orientation of a stream. (A successful call to **freopen** removes any orientation.)[224)]

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

— Binary wide-oriented streams have the file-positioning restrictions ascribed to both text and binary streams.

— For wide-oriented streams, after a successful call to a file-positioning function that leaves the file position indicator prior to the end-of-file, a wide character output function can overwrite a partial multibyte character; any file contents beyond the byte(s) written are henceforth indeterminate.

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

Environmental limits

7 An implementation shall support text files with lines containing at least 254 characters, including the terminating new-line character. The value of the macro **BUFSIZ** shall be at least 256.

Forward references: the **freopen** function (7.19.5.4), the **fwide** function (7.24.3.5), **mbstate_t** (7.25.1), the **fgetpos** function (7.19.9.1), the **fsetpos** function (7.19.9.3).

224) The three predefined streams **stdin**, **stdout**, and **stderr** are unoriented at program startup.

7.19.3 Files

1 A stream is associated with an external file (which may be a physical device) by *opening* a file, which may involve *creating* a new file. Creating an existing file causes its former contents to be discarded, if necessary. If a file can support positioning requests (such as a disk file, as opposed to a terminal), then a *file position indicator* associated with the stream is positioned at the start (character number zero) of the file, unless the file is opened with append mode in which case it is implementation-defined whether the file position indicator is initially positioned at the beginning or the end of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file.

2 Binary files are not truncated, except as defined in 7.19.5.3. Whether a write on a text stream causes the associated file to be truncated beyond that point is implementation-defined.

3 When a stream is *unbuffered*, characters are intended to appear from the source or at the destination as soon as possible. Otherwise characters may be accumulated and transmitted to or from the host environment as a block. When a stream is *fully buffered*, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a stream is *line buffered*, characters are intended to be transmitted to or from the host environment as a block when a new-line character is encountered. Furthermore, characters are intended to be transmitted as a block to the host environment when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line buffered stream that requires the transmission of characters from the host environment. Support for these characteristics is implementation-defined, and may be affected via the **setbuf** and **setvbuf** functions.

4 A file may be disassociated from a controlling stream by *closing* the file. Output streams are flushed (any unwritten buffer contents are transmitted to the host environment) before the stream is disassociated from the file. The value of a pointer to a **FILE** object is indeterminate after the associated file is closed (including the standard text streams). Whether a file of zero length (on which no characters have been written by an output stream) actually exists is implementation-defined.

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

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

7 At program startup, three text streams are predefined and need not be opened explicitly — *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). As initially opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

8 Functions that open additional (nontemporary) files require a *file name*, which is a string. The rules for composing valid file names are implementation-defined. Whether the same file can be simultaneously open multiple times is also implementation-defined.

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

— Multibyte encodings within files may contain embedded null bytes (unlike multibyte encodings valid for use internal to the program).

— A file need not begin nor end in the initial shift state.[225)]

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

11 The wide character input functions read multibyte characters from the stream and convert them to wide characters as if they were read by successive calls to the **fgetwc** function. Each conversion occurs as if by a call to the **mbrtowc** function, with the conversion state described by the stream's own **mbstate_t** object. The byte input functions read characters from the stream as if by successive calls to the **fgetc** function.

12 The wide character output functions convert wide characters to multibyte characters and write them to the stream as if they were written by successive calls to the **fputwc** function. Each conversion occurs as if by a call to the **wcrtomb** function, with the conversion state described by the stream's own **mbstate_t** object. The byte output functions write characters to the stream as if by successive calls to the **fputc** function.

13 In some cases, some of the byte input/output functions also perform conversions between multibyte characters and wide characters. These conversions also occur as if by calls to the **mbrtowc** and **wcrtomb** functions.

14 An *encoding error* occurs if the character sequence presented to the underlying **mbrtowc** function does not form a valid (generalized) multibyte character, or if the code value passed to the underlying **wcrtomb** does not correspond to a valid (generalized)

225) Setting the file position indicator to end-of-file, as with **fseek(file, 0, SEEK_END)**, has undefined behavior for a binary stream (because of possible trailing null characters) or for any stream with state-dependent encoding that does not assuredly end in the initial shift state.

multibyte character. The wide character input/output functions and the byte input/output functions store the value of the macro **EILSEQ** in **errno** if and only if an encoding error occurs.

Environmental limits

15 The value of **FOPEN_MAX** shall be at least eight, including the three standard text streams.

Forward references: the **exit** function (7.20.4.3), the **fgetc** function (7.19.7.1), the **fopen** function (7.19.5.3), the **fputc** function (7.19.7.3), the **setbuf** function (7.19.5.5), the **setvbuf** function (7.19.5.6), the **fgetwc** function (7.24.3.1), the **fputwc** function (7.24.3.3), conversion state (7.24.6), the **mbrtowc** function (7.24.6.3.2), the **wcrtomb** function (7.24.6.3.3).

7.19.4 Operations on files

7.19.4.1 The remove function

Synopsis

1
```
#include <stdio.h>
int remove(const char *filename);
```

Description

2 The **remove** function causes the file whose name is the string pointed to by **filename** to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew. If the file is open, the behavior of the **remove** function is implementation-defined.

Returns

3 The **remove** function returns zero if the operation succeeds, nonzero if it fails.

7.19.4.2 The rename function

Synopsis

1
```
#include <stdio.h>
int rename(const char *old, const char *new);
```

Description

2 The **rename** function causes the file whose name is the string pointed to by **old** to be henceforth known by the name given by the string pointed to by **new**. The file named **old** is no longer accessible by that name. If a file named by the string pointed to by **new** exists prior to the call to the **rename** function, the behavior is implementation-defined.

Returns

3 The **rename** function returns zero if the operation succeeds, nonzero if it fails,[226] in which case if the file existed previously it is still known by its original name.

7.19.4.3 The tmpfile function

Synopsis

1
```
#include <stdio.h>
FILE *tmpfile(void);
```

Description

2 The **tmpfile** function creates a temporary binary file that is different from any other existing file and that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with **"wb+"** mode.

Recommended practice

It should be possible to open at least **TMP_MAX** temporary files during the lifetime of the program (this limit may be shared with **tmpnam**) and there should be no limit on the number simultaneously open other than this limit and any limit on the number of open files (**FOPEN_MAX**).

Returns

3 The **tmpfile** function returns a pointer to the stream of the file that it created. If the file cannot be created, the **tmpfile** function returns a null pointer.

Forward references: the **fopen** function (7.19.5.3).

7.19.4.4 The tmpnam function

Synopsis

1
```
#include <stdio.h>
char *tmpnam(char *s);
```

Description

2 The **tmpnam** function generates a string that is a valid file name and that is not the same as the name of an existing file.[227]

---

226) Among the reasons the implementation may cause the **rename** function to fail are that the file is open or that it is necessary to copy its contents to effectuate its renaming.

227) Files created using strings generated by the **tmpnam** function are temporary only in the sense that their names should not collide with those generated by conventional naming rules for the implementation. It is still necessary to use the **remove** function to remove such files when their use is ended, and before program termination.

3 The **tmpnam** function generates a different string each time it is called. The function is capable of generating **TMP_MAX** different strings, but any or all of them may already be in use by existing files.

4 The implementation shall behave as if no library function calls the **tmpnam** function.

Returns

5 If no suitable string can be generated, the **tmpnam** function returns a null pointer. Otherwise, if the argument is a null pointer, the **tmpnam** function leaves its result in an internal static object and returns a pointer to that object (subsequent calls to the **tmpnam** function may modify the same object). If the argument is not a null pointer, it is assumed to point to an array of at least **L_tmpnam** **char**s; the **tmpnam** function writes its result in that array and returns the argument as its value.

Environmental limits

6 The value of the macro **TMP_MAX** shall be at least 25.

7.19.5 File access functions

7.19.5.1 The fclose function

Synopsis

1
```
#include <stdio.h>
int fclose(FILE *stream);
```

Description

2 A successful call to the **fclose** function causes the stream pointed to by **stream** to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; any unread buffered data are discarded. Whether or not the call succeeds, the stream is disassociated from the file and any buffer set by the **setbuf** or **setvbuf** function is disassociated from the stream (and deallocated if it was automatically allocated).

Returns

3 The **fclose** function returns zero if the stream was successfully closed, or **EOF** if any errors were detected.

7.19.5.2 The fflush function

Synopsis

1
```
#include <stdio.h>
int fflush(FILE *stream);
```

Description

2 If **stream** points to an output stream or an update stream in which the most recent operation was not input, the **fflush** function causes any unwritten data for that stream to be delivered to the host environment to be written to the file; otherwise, the behavior is undefined.

3 If **stream** is a null pointer, the **fflush** function performs this flushing action on all streams for which the behavior is defined above.

Returns

4 The **fflush** function sets the error indicator for the stream and returns **EOF** if a write error occurs, otherwise it returns zero.

Forward references: the **fopen** function (7.19.5.3).

7.19.5.3 The fopen function

Synopsis

1
```
#include <stdio.h>
FILE *fopen(const char * restrict filename,
     const char * restrict mode);
```

Description

2 The **fopen** function opens the file whose name is the string pointed to by **filename**, and associates a stream with it.

3 The argument **mode** points to a string. If the string is one of the following, the file is open in the indicated mode. Otherwise, the behavior is undefined.[228)]

r	open text file for reading
w	truncate to zero length or create text file for writing
a	append; open or create text file for writing at end-of-file
rb	open binary file for reading
wb	truncate to zero length or create binary file for writing
ab	append; open or create binary file for writing at end-of-file
r+	open text file for update (reading and writing)
w+	truncate to zero length or create text file for update
a+	append; open or create text file for update, writing at end-of-file

228) If the string begins with one of the above sequences, the implementation might choose to ignore the remaining characters, or it might use them to select different kinds of a file (some of which might not conform to the properties in 7.19.2).

r+b *or* **rb+** open binary file for update (reading and writing)
w+b *or* **wb+** truncate to zero length or create binary file for update
a+b *or* **ab+** append; open or create binary file for update, writing at end-of-file

4 Opening a file with read mode (**'r'** as the first character in the **mode** argument) fails if the file does not exist or cannot be read.

5 Opening a file with append mode (**'a'** as the first character in the **mode** argument) causes all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to the **fseek** function. In some implementations, opening a binary file with append mode (**'b'** as the second or third character in the above list of **mode** argument values) may initially position the file position indicator for the stream beyond the last data written, because of null character padding.

6 When a file is opened with update mode (**'+'** as the second or third character in the above list of **mode** argument values), both input and output may be performed on the associated stream. However, output shall not be directly followed by input without an intervening call to the **fflush** function or to a file positioning function (**fseek**, **fsetpos**, or **rewind**), and input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations.

7 When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

Returns

8 The **fopen** function returns a pointer to the object controlling the stream. If the open operation fails, **fopen** returns a null pointer.

Forward references: file positioning functions (7.19.9).

7.19.5.4 The freopen function

Synopsis

1
```
#include <stdio.h>
FILE *freopen(const char * restrict filename,
     const char * restrict mode,
     FILE * restrict stream);
```

Description

2 The **freopen** function opens the file whose name is the string pointed to by **filename** and associates the stream pointed to by **stream** with it. The **mode** argument is used just

as in the **fopen** function.[229)]

3 If **filename** is a null pointer, the **freopen** function attempts to change the mode of the stream to that specified by **mode**, as if the name of the file currently associated with the stream had been used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

4 The **freopen** function first attempts to close any file that is associated with the specified stream. Failure to close the file is ignored. The error and end-of-file indicators for the stream are cleared.

Returns

5 The **freopen** function returns a null pointer if the open operation fails. Otherwise, **freopen** returns the value of **stream**.

7.19.5.5 The setbuf function

Synopsis

1
```
#include <stdio.h>
void setbuf(FILE * restrict stream,
     char * restrict buf);
```

Description

2 Except that it returns no value, the **setbuf** function is equivalent to the **setvbuf** function invoked with the values **_IOFBF** for **mode** and **BUFSIZ** for **size**, or (if **buf** is a null pointer), with the value **_IONBF** for **mode**.

Returns

3 The **setbuf** function returns no value.

Forward references: the **setvbuf** function (7.19.5.6).

7.19.5.6 The setvbuf function

Synopsis

1
```
#include <stdio.h>
int setvbuf(FILE * restrict stream,
     char * restrict buf,
     int mode, size_t size);
```

229) The primary use of the **freopen** function is to change the file associated with a standard text stream (**stderr**, **stdin**, or **stdout**), as those identifiers need not be modifiable lvalues to which the value returned by the **fopen** function may be assigned.

Description

2 The **setvbuf** function may be used only after the stream pointed to by **stream** has been associated with an open file and before any other operation (other than an unsuccessful call to **setvbuf**) is performed on the stream. The argument **mode** determines how **stream** will be buffered, as follows: **_IOFBF** causes input/output to be fully buffered; **_IOLBF** causes input/output to be line buffered; **_IONBF** causes input/output to be unbuffered. If **buf** is not a null pointer, the array it points to may be used instead of a buffer allocated by the **setvbuf** function[230] and the argument **size** specifies the size of the array; otherwise, **size** may determine the size of a buffer allocated by the **setvbuf** function. The contents of the array at any time are indeterminate.

Returns

3 The **setvbuf** function returns zero on success, or nonzero if an invalid value is given for **mode** or if the request cannot be honored.

7.19.6 Formatted input/output functions

1 The formatted input/output functions shall behave as if there is a sequence point after the actions associated with each specifier.[231]

7.19.6.1 The fprintf function

Synopsis

1
```
#include <stdio.h>
int fprintf(FILE * restrict stream,
     const char * restrict format, ...);
```

Description

2 The **fprintf** function writes output to the stream pointed to by **stream**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The **fprintf** function returns when the end of the format string is encountered.

3 The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not **%**), which are copied unchanged to the output stream; and conversion

230) The buffer has to have a lifetime at least as great as the open stream, so the stream should be closed before a buffer that has automatic storage duration is deallocated upon block exit.

231) The **fprintf** functions perform writes to memory for the **%n** specifier.

specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

4 Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

— Zero or more *flags* (in any order) that modify the meaning of the conversion specification.

— An optional minimum *field width*. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk `*` (described later) or a decimal integer.[232)]

— An optional *precision* that gives the minimum number of digits to appear for the **`d`**, **`i`**, **`o`**, **`u`**, **`x`**, and **`X`** conversions, the number of digits to appear after the decimal-point character for **`a`**, **`A`**, **`e`**, **`E`**, **`f`**, and **`F`** conversions, the maximum number of significant digits for the **`g`** and **`G`** conversions, or the maximum number of bytes to be written for **`s`** conversions. The precision takes the form of a period (`.`) followed either by an asterisk `*` (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

— An optional *length modifier* that specifies the size of the argument.

— A *conversion specifier* character that specifies the type of conversion to be applied.

5 As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an **`int`** argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a **`-`** flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

6 The flag characters and their meanings are:

`-` The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)

`+` The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not

232) Note that **`0`** is taken as a flag, not as the beginning of a field width.

specified.)[233)]

space If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the *space* and **+** flags both appear, the *space* flag is ignored.

The result is converted to an "alternative form". For **o** conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For **x** (or **X**) conversion, a nonzero result has **0x** (or **0X**) prefixed to it. For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For **g** and **G** conversions, trailing zeros are *not* removed from the result. For other conversions, the behavior is undefined.

0 For **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the **0** and **-** flags both appear, the **0** flag is ignored. For **d**, **i**, **o**, **u**, **x**, and **X** conversions, if a precision is specified, the **0** flag is ignored. For other conversions, the behavior is undefined.

7 The length modifiers and their meanings are:

hh Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **signed char** or **unsigned char** before printing); or that a following **n** conversion specifier applies to a pointer to a **signed char** argument.

h Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **short int** or **unsigned short int** before printing); or that a following **n** conversion specifier applies to a pointer to a **short int** argument.

l (ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long int** or **unsigned long int** argument; that a following **n** conversion specifier applies to a pointer to a **long int** argument; that a

233) The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.

following **c** conversion specifier applies to a **wint_t** argument; that a following **s** conversion specifier applies to a pointer to a **wchar_t** argument; or has no effect on a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier.

ll (ell-ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long long int** or **unsigned long long int** argument; or that a following **n** conversion specifier applies to a pointer to a **long long int** argument.

j Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to an **intmax_t** or **uintmax_t** argument; or that a following **n** conversion specifier applies to a pointer to an **intmax_t** argument.

z Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **size_t** or the corresponding signed integer type argument; or that a following **n** conversion specifier applies to a pointer to a signed integer type corresponding to **size_t** argument.

t Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **ptrdiff_t** or the corresponding unsigned integer type argument; or that a following **n** conversion specifier applies to a pointer to a **ptrdiff_t** argument.

L Specifies that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to a **long double** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

8 The conversion specifiers and their meanings are:

d,i The **int** argument is converted to signed decimal in the style *[−]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

o,u,x,X The **unsigned int** argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X**) in the style *dddd*; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f,F A **double** argument representing a floating-point number is converted to decimal notation in the style *[−]ddd*.*ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

A **double** argument representing an infinity is converted in one of the styles *[−]***inf** or *[−]***infinity** — which style is implementation-defined. A **double** argument representing a NaN is converted in one of the styles *[−]***nan** or *[−]***nan(***n-char-sequence***)** — which style, and the meaning of any *n-char-sequence*, is implementation-defined. The **F** conversion specifier produces **INF**, **INFINITY**, or **NAN** instead of **inf**, **infinity**, or **nan**, respectively.[234)]

e,E A **double** argument representing a floating-point number is converted in the style *[−]d*.*ddd* **e**±*dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier produces a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.

A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

g,G A **double** argument representing a floating-point number is converted in style **f** or **e** (or in style **F** or **E** in the case of a **G** conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style **e** (or **E**) is used only if the exponent resulting from such a conversion is less than −4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result unless the **#** flag is specified; a decimal-point character appears only if it is followed by a digit.

A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

---

234) When applied to infinite and NaN values, the **−**, **+**, and *space* flag characters have their usual meaning; the **#** and **0** flag characters have no effect.

a,A A **double** argument representing a floating-point number is converted in the style *[−]***0x***h*.*hhhh***p**±*d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character[235] and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and **FLT_RADIX** is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and **FLT_RADIX** is not a power of 2, then the precision is sufficient to distinguish[236] values of type **double**, except that trailing zeros may be omitted; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. The letters **abcdef** are used for **a** conversion and the letters **ABCDEF** for **A** conversion. The **A** conversion specifier produces a number with **X** and **P** instead of **x** and **p**. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

c If no **l** length modifier is present, the **int** argument is converted to an **unsigned char**, and the resulting character is written.

If an **l** length modifier is present, the **wint_t** argument is converted as if by an **ls** conversion specification with no precision and an argument that points to the initial element of a two-element array of **wchar_t**, the first element containing the **wint_t** argument to the **lc** conversion specification and the second a null wide character.

s If no **l** length modifier is present, the argument shall be a pointer to the initial element of an array of character type.[237] Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many bytes are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

If an **l** length modifier is present, the argument shall be a pointer to the initial

---

235) Binary implementations can choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble (4-bit) boundaries.

236) The precision p is sufficient to distinguish values of the source type if $16^{p-1} > b^n$ where b is **FLT_RADIX** and n is the number of base-b digits in the significand of the source type. A smaller p might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point character.

237) No special provisions are made for multibyte characters.

element of an array of **wchar_t** type. Wide characters from the array are converted to multibyte characters (each as if by a call to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many bytes are written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the multibyte character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case is a partial multibyte character written.[238)]

p The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printing characters, in an implementation-defined manner.

n The argument shall be a pointer to signed integer into which is *written* the number of characters written to the output stream so far by this call to **fprintf**. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

% A **%** character is written. No argument is converted. The complete conversion specification shall be **%%**.

9 If a conversion specification is invalid, the behavior is undefined.[239)] If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

10 In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

11 For **a** and **A** conversions, if **FLT_RADIX** is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

Recommended practice

12 If **FLT_RADIX** is not a power of 2, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

---

238) Redundant shift sequences may result if multibyte characters have a state-dependent encoding.

239) See ‘‘future library directions’’ (7.26.9).

13 For **e**, **E**, **f**, **F**, **g**, and **G** conversions, if the number of significant decimal digits is at most **DECIMAL_DIG**, then the result should be correctly rounded.[240] If the number of significant decimal digits is more than **DECIMAL_DIG** but the source value is exactly representable with **DECIMAL_DIG** digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having **DECIMAL_DIG** significant digits; the value of the resultant decimal string D should satisfy $L \le D \le U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

Returns

14 The **fprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

Environmental limits

15 The number of characters that can be produced by any single conversion shall be at least 4095.

16 EXAMPLE 1 To print a date and time in the form ‘‘Sunday, July 3, 10:02’’ followed by π to five decimal places:

```
#include <math.h>
#include <stdio.h>
/* ... */
char *weekday, *month;     // pointers to strings
int day, hour, min;
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
      weekday, month, day, hour, min);
fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));
```

17 EXAMPLE 2 In this example, multibyte characters do not have a state-dependent encoding, and the members of the extended character set that consist of more than one byte each consist of exactly two bytes, the first of which is denoted here by a □ and the second by an uppercase letter.

18 Given the following wide string with length seven,

```
static wchar_t wstr[] = L"□X□Yabc□Z□W";
```

the seven calls

```
fprintf(stdout, "|1234567890123|\n");
fprintf(stdout, "|%13ls|\n", wstr);
fprintf(stdout, "|%-13.9ls|\n", wstr);
fprintf(stdout, "|%13.10ls|\n", wstr);
fprintf(stdout, "|%13.11ls|\n", wstr);
fprintf(stdout, "|%13.15ls|\n", &wstr[2]);
fprintf(stdout, "|%13lc|\n", wstr[5]);
```

240) For binary-to-decimal conversion, the result format’s values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.

will print the following seven lines:

```
|1234567890123|
|   □X□Yabc□Z□W|
|□X□Yabc□Z     |
|     □X□Yabc□Z|
|   □X□Yabc□Z□W|
|       abc□Z□W|
|            □Z|
```

Forward references: conversion state (7.24.6), the **wcrtomb** function (7.24.6.3.3).

7.19.6.2 The fscanf function

Synopsis

1
```
#include <stdio.h>
int fscanf(FILE * restrict stream,
     const char * restrict format, ...);
```

Description

2 The **fscanf** function reads input from the stream pointed to by **stream**, under control of the string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

3 The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters, an ordinary multibyte character (neither `%` nor a white-space character), or a conversion specification. Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

— An optional assignment-suppressing character `*`.

— An optional nonzero decimal integer that specifies the maximum field width (in characters).

— An optional *length modifier* that specifies the size of the receiving object.

— A *conversion specifier* character that specifies the type of conversion to be applied.

4 The **fscanf** function executes each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the occurrence of an encoding error or the unavailability of input characters), or matching failures (due to inappropriate input).

5 A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can

be read.

6 A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If any of those characters differ from the ones composing the directive, the directive fails and the differing and subsequent characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive fails.

7 A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

8 Input white-space characters (as specified by the **isspace** function) are skipped, unless the specification includes a **[**, **c**, or **n** specifier.[241)]

9 An input item is read from the stream, unless the specification includes an **n** specifier. An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence.[242)] The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

10 Except in the case of a **%** specifier, the input item (or, in the case of a **%n** directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *****, the result of the conversion is placed in the object pointed to by the first argument following the **format** argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

11 The length modifiers and their meanings are:

hh Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.

h Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **short int** or **unsigned short int**.

241) These white-space characters are not counted against a specified field width.

242) **fscanf** pushes back at most one input character onto the input stream. Therefore, some sequences that are acceptable to **strtod**, **strtol**, etc., are unacceptable to **fscanf**.

l (ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **long int** or **unsigned long int**; that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to an argument with type pointer to **double**; or that a following **c**, **s**, or **[** conversion specifier applies to an argument with type pointer to **wchar_t**.

ll (ell-ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **long long int** or **unsigned long long int**.

j Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **intmax_t** or **uintmax_t**.

z Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **size_t** or the corresponding signed integer type.

t Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **ptrdiff_t** or the corresponding unsigned integer type.

L Specifies that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to an argument with type pointer to **long double**.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

12 The conversion specifiers and their meanings are:

d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to signed integer.

i Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 0 for the **base** argument. The corresponding argument shall be a pointer to signed integer.

o Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 8 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

u Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

x Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 16 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

a,e,f,g Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of the **strtod** function. The corresponding argument shall be a pointer to floating.

c Matches a sequence of characters of exactly the number specified by the field width (1 if no field width is present in the directive).[243)]

If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character in the sequence is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the resulting sequence of wide characters. No null wide character is added.

s Matches a sequence of non-white-space characters.[243)]

If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

---

243) No special provisions are made for multibyte characters in the matching rules used by the **c**, **s**, and **[** conversion specifiers — the extent of the input field is determined on a byte-by-byte basis. The resulting field is nevertheless a sequence of multibyte characters that begins in the initial shift state.

[Matches a nonempty sequence of characters from a set of expected characters (the *scanset*).[243)]

If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

The conversion specifier includes all subsequent characters in the **format** string, up to and including the matching right bracket (**]**). The characters between the brackets (the *scanlist*) compose the scanset, unless the character after the left bracket is a circumflex (**^**), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with **[]** or **[^]**, the right bracket character is in the scanlist and the next following right bracket character is the matching right bracket that ends the specification; otherwise the first following right bracket character is the one that ends the specification. If a **-** character is in the scanlist and is not the first, nor the second where the first character is a **^**, nor the last character, the behavior is implementation-defined.

p Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the **%p** conversion of the **fprintf** function. The corresponding argument shall be a pointer to a pointer to **void**. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the **%p** conversion is undefined.

n No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of characters read from the input stream so far by this call to the **fscanf** function. Execution of a **%n** directive does not increment the assignment count returned at the completion of execution of the **fscanf** function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

`%` Matches a single `%` character; no conversion or assignment occurs. The complete conversion specification shall be `%%`.

13 If a conversion specification is invalid, the behavior is undefined.[244)]

14 The conversion specifiers **A**, **E**, **F**, **G**, and **X** are also valid and behave the same as, respectively, **a**, **e**, **f**, **g**, and **x**.

15 Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the **%n** directive.

Returns

16 The **fscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

17 EXAMPLE 1 The call:

```
#include <stdio.h>
/* ... */
int n, i; float x; char name[50];
n = fscanf(stdin, "%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to **n** the value 3, to **i** the value 25, to **x** the value 5.432, and to **name** the sequence **thompson\0**.

18 EXAMPLE 2 The call:

```
#include <stdio.h>
/* ... */
int i; float x; char name[50];
fscanf(stdin, "%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign to **i** the value 56 and to **x** the value 789.0, will skip **0123**, and will assign to **name** the sequence **56\0**. The next character read from the input stream will be **a**.

244) See ‘‘future library directions’’ (7.26.9).

19 EXAMPLE 3 To accept repeatedly from **stdin** a quantity, a unit of measure, and an item name:

```
#include <stdio.h>
/* ... */
int count; float quant; char units[21], item[21];
do {
      count = fscanf(stdin, "%f%20s of %20s", &quant, units, item);
      fscanf(stdin,"%*[^\n]");
} while (!feof(stdin) && !ferror(stdin));
```

20 If the **stdin** stream contains the following lines:

```
2 quarts of oil
-12.8degrees Celsius
lots of luck
10.0LBS     of
dirt
100ergs of energy
```

the execution of the above example will be analogous to the following assignments:

```
quant = 2; strcpy(units, "quarts"); strcpy(item, "oil");
count = 3;
quant = -12.8; strcpy(units, "degrees");
count = 2; // "C" fails to match "o"
count = 0; // "l" fails to match "%f"
quant = 10.0; strcpy(units, "LBS"); strcpy(item, "dirt");
count = 3;
count = 0; // "100e" fails to match "%f"
count = EOF;
```

21 EXAMPLE 4 In:

```
#include <stdio.h>
/* ... */
int d1, d2, n1, n2, i;
i = sscanf("123", "%d%n%n%d", &d1, &n1, &n2, &d2);
```

the value 123 is assigned to **d1** and the value 3 to **n1**. Because **%n** can never get an input failure the value of 3 is also assigned to **n2**. The value of **d2** is not affected. The value 1 is assigned to **i**.

22 EXAMPLE 5 In these examples, multibyte characters do have a state-dependent encoding, and the members of the extended character set that consist of more than one byte each consist of exactly two bytes, the first of which is denoted here by a □ and the second by an uppercase letter, but are only recognized as such when in the alternate shift state. The shift sequences are denoted by ↑ and ↓, in which the first causes entry into the alternate shift state.

23 After the call:

```
#include <stdio.h>
/* ... */
char str[50];
fscanf(stdin, "a%s", str);
```

with the input line:

```
a↑□X□Y↓ bc
```

str will contain ↑□**X**□**Y**↓**\0** assuming that none of the bytes of the shift sequences (or of the multibyte characters, in the more general case) appears to be a single-byte white-space character.

24 In contrast, after the call:

```
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a%ls", wstr);
```

with the same input line, **wstr** will contain the two wide characters that correspond to □**X** and □**Y** and a terminating null wide character.

25 However, the call:

```
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a↑□X↓%ls", wstr);
```

with the same input line will return zero due to a matching failure against the ↓ sequence in the format string.

26 Assuming that the first byte of the multibyte character □**X** is the same as the first byte of the multibyte character □**Y**, after the call:

```
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a↑□Y↓%ls", wstr);
```

with the same input line, zero will again be returned, but **stdin** will be left with a partially consumed multibyte character.

Forward references: the **strtod**, **strtof**, and **strtold** functions (7.20.1.3), the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions (7.20.1.4), conversion state (7.24.6), the **wcrtomb** function (7.24.6.3.3).

7.19.6.3 The printf function

Synopsis

1
```
#include <stdio.h>
int printf(const char * restrict format, ...);
```

Description

2 The **printf** function is equivalent to **fprintf** with the argument **stdout** interposed before the arguments to **printf**.

Returns

3 The **printf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

7.19.6.4 The `scanf` function

Synopsis

1
```
#include <stdio.h>
int scanf(const char * restrict format, ...);
```

Description

2 The **scanf** function is equivalent to **fscanf** with the argument **stdin** interposed before the arguments to **scanf**.

Returns

3 The **scanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **scanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.19.6.5 The `snprintf` function

Synopsis

1
```
#include <stdio.h>
int snprintf(char * restrict s, size_t n,
     const char * restrict format, ...);
```

Description

2 The **snprintf** function is equivalent to **fprintf**, except that the output is written into an array (specified by argument **s**) rather than to a stream. If **n** is zero, nothing is written, and **s** may be a null pointer. Otherwise, output characters beyond the **n-1**st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **snprintf** function returns the number of characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**.

7.19.6.6 The `sprintf` function

Synopsis

1
```
#include <stdio.h>
int sprintf(char * restrict s,
     const char * restrict format, ...);
```

Description

2 The **sprintf** function is equivalent to **fprintf**, except that the output is written into an array (specified by the argument **s**) rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **sprintf** function returns the number of characters written in the array, not counting the terminating null character, or a negative value if an encoding error occurred.

7.19.6.7 The sscanf function

Synopsis

1
```
#include <stdio.h>
int sscanf(const char * restrict s,
     const char * restrict format, ...);
```

Description

2 The **sscanf** function is equivalent to **fscanf**, except that input is obtained from a string (specified by the argument **s**) rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the **fscanf** function. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **sscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **sscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.19.6.8 The vfprintf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE * restrict stream,
     const char * restrict format,
     va_list arg);
```

Description

2 The **vfprintf** function is equivalent to **fprintf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfprintf** function does not invoke the

va_end macro.[245)]

Returns

3 The **vfprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

4 EXAMPLE The following shows the use of the **vfprintf** function in a general error-reporting routine.

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

void error(char *function_name, char *format, ...)
{
      va_list args;

      va_start(args, format);
      // print out name of function causing error
      fprintf(stderr, "ERROR in %s: ", function_name);
      // print out remainder of message
      vfprintf(stderr, format, args);
      va_end(args);
}
```

7.19.6.9 The vfscanf function

Synopsis

1

```
#include <stdarg.h>
#include <stdio.h>
int vfscanf(FILE * restrict stream,
     const char * restrict format,
     va_list arg);
```

Description

2 The **vfscanf** function is equivalent to **fscanf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfscanf** function does not invoke the **va_end** macro.[245)]

Returns

3 The **vfscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **vfscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

245) As the functions **vfprintf**, **vfscanf**, **vprintf**, **vscanf**, **vsnprintf**, **vsprintf**, and **vsscanf** invoke the **va_arg** macro, the value of **arg** after the return is indeterminate.

7.19.6.10 The `vprintf` function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char * restrict format,
     va_list arg);
```

Description

2 The **vprintf** function is equivalent to **printf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vprintf** function does not invoke the **va_end** macro.[245)]

Returns

3 The **vprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

7.19.6.11 The `vscanf` function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vscanf(const char * restrict format,
     va_list arg);
```

Description

2 The **vscanf** function is equivalent to **scanf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vscanf** function does not invoke the **va_end** macro.[245)]

Returns

3 The **vscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **vscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.19.6.12 The `vsnprintf` function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char * restrict s, size_t n,
     const char * restrict format,
     va_list arg);
```

Description

2 The **vsnprintf** function is equivalent to **snprintf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsnprintf** function does not invoke the **va_end** macro.[245)] If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **vsnprintf** function returns the number of characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**.

7.19.6.13 The `vsprintf` function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char * restrict s,
     const char * restrict format,
     va_list arg);
```

Description

2 The **vsprintf** function is equivalent to **sprintf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsprintf** function does not invoke the **va_end** macro.[245)] If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **vsprintf** function returns the number of characters written in the array, not counting the terminating null character, or a negative value if an encoding error occurred.

7.19.6.14 The `vsscanf` function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vsscanf(const char * restrict s,
     const char * restrict format,
     va_list arg);
```

Description

2 The **vsscanf** function is equivalent to **sscanf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsscanf** function does not invoke the **va_end** macro.[245)]

Returns

3 The **vsscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **vscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.19.7 Character input/output functions

7.19.7.1 The `fgetc` function

Synopsis

1
```
#include <stdio.h>
int fgetc(FILE *stream);
```

Description

2 If the end-of-file indicator for the input stream pointed to by **stream** is not set and a next character is present, the **fgetc** function obtains that character as an **unsigned char** converted to an **int** and advances the associated file position indicator for the stream (if defined).

Returns

3 If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for the stream is set and **fgetc** returns **EOF**. Otherwise, the **fgetc** function returns the next character from the input stream pointed to by **stream**. If a read error occurs, the error indicator for the stream is set and **fgetc** returns **EOF**.[246)]

246) An end-of-file and a read error can be distinguished by use of the **feof** and **ferror** functions.

7.19.7.2 The `fgets` function

Synopsis

1
```
#include <stdio.h>
char *fgets(char * restrict s, int n,
     FILE * restrict stream);
```

Description

2 The **fgets** function reads at most one less than the number of characters specified by **n** from the stream pointed to by **stream** into the array pointed to by **s**. No additional characters are read after a new-line character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

Returns

3 The **fgets** function returns **s** if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

7.19.7.3 The `fputc` function

Synopsis

1
```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

Description

2 The **fputc** function writes the character specified by **c** (converted to an **unsigned char**) to the output stream pointed to by **stream**, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

Returns

3 The **fputc** function returns the character written. If a write error occurs, the error indicator for the stream is set and **fputc** returns **EOF**.

7.19.7.4 The `fputs` function

Synopsis

1
```
#include <stdio.h>
int fputs(const char * restrict s,
     FILE * restrict stream);
```

Description

2 The **fputs** function writes the string pointed to by **s** to the stream pointed to by **stream**. The terminating null character is not written.

Returns

3 The **fputs** function returns **EOF** if a write error occurs; otherwise it returns a nonnegative value.

7.19.7.5 The getc function

Synopsis

1
```
#include <stdio.h>
int getc(FILE *stream);
```

Description

2 The **getc** function is equivalent to **fgetc**, except that if it is implemented as a macro, it may evaluate **stream** more than once, so the argument should never be an expression with side effects.

Returns

3 The **getc** function returns the next character from the input stream pointed to by **stream**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getc** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **getc** returns **EOF**.

7.19.7.6 The getchar function

Synopsis

1
```
#include <stdio.h>
int getchar(void);
```

Description

2 The **getchar** function is equivalent to **getc** with the argument **stdin**.

Returns

3 The **getchar** function returns the next character from the input stream pointed to by **stdin**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getchar** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **getchar** returns **EOF**.

7.19.7.7 The `gets` function

Synopsis

1
```
#include <stdio.h>
char *gets(char *s);
```

Description

2 The **gets** function reads characters from the input stream pointed to by **stdin**, into the array pointed to by **s**, until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

Returns

3 The **gets** function returns **s** if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

7.19.7.8 The `putc` function

Synopsis

1
```
#include <stdio.h>
int putc(int c, FILE *stream);
```

Description

2 The **putc** function is equivalent to **fputc**, except that if it is implemented as a macro, it may evaluate **stream** more than once, so that argument should never be an expression with side effects.

Returns

3 The **putc** function returns the character written. If a write error occurs, the error indicator for the stream is set and **putc** returns **EOF**.

7.19.7.9 The `putchar` function

Synopsis

1
```
#include <stdio.h>
int putchar(int c);
```

Description

2 The **putchar** function is equivalent to **putc** with the second argument **stdout**.

Returns

3 The **putchar** function returns the character written. If a write error occurs, the error indicator for the stream is set and **putchar** returns **EOF**.

7.19.7.10 The puts function

Synopsis

1
```
#include <stdio.h>
int puts(const char *s);
```

Description

2 The **puts** function writes the string pointed to by **s** to the stream pointed to by **stdout**, and appends a new-line character to the output. The terminating null character is not written.

Returns

3 The **puts** function returns **EOF** if a write error occurs; otherwise it returns a nonnegative value.

7.19.7.11 The ungetc function

Synopsis

1
```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Description

2 The **ungetc** function pushes the character specified by **c** (converted to an **unsigned char**) back onto the input stream pointed to by **stream**. Pushed-back characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by **stream**) to a file positioning function (**fseek**, **fsetpos**, or **rewind**) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

3 One character of pushback is guaranteed. If the **ungetc** function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

4 If the value of **c** equals that of the macro **EOF**, the operation fails and the input stream is unchanged.

5 A successful call to the **ungetc** function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back. For a text stream, the value of its file position indicator after a successful call to the **ungetc** function is unspecified until all pushed-back characters are read or discarded. For a binary stream, its file position indicator is decremented by each successful call to the **ungetc** function; if its value was zero before a call, it is indeterminate after the

call.[247)]

Returns

6 The **ungetc** function returns the character pushed back after conversion, or **EOF** if the operation fails.

Forward references: file positioning functions (7.19.9).

7.19.8 Direct input/output functions

7.19.8.1 The fread function

Synopsis

1
```
#include <stdio.h>
size_t fread(void * restrict ptr,
     size_t size, size_t nmemb,
     FILE * restrict stream);
```

Description

2 The **fread** function reads, into the array pointed to by **ptr**, up to **nmemb** elements whose size is specified by **size**, from the stream pointed to by **stream**. For each object, **size** calls are made to the **fgetc** function and the results stored, in the order read, in an array of **unsigned char** exactly overlaying the object. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

Returns

3 The **fread** function returns the number of elements successfully read, which may be less than **nmemb** if a read error or end-of-file is encountered. If **size** or **nmemb** is zero, **fread** returns zero and the contents of the array and the state of the stream remain unchanged.

7.19.8.2 The fwrite function

Synopsis

1
```
#include <stdio.h>
size_t fwrite(const void * restrict ptr,
     size_t size, size_t nmemb,
     FILE * restrict stream);
```

247) See ‘‘future library directions’’ (7.26.9).

Description

2 The **fwrite** function writes, from the array pointed to by **ptr**, up to **nmemb** elements whose size is specified by **size**, to the stream pointed to by **stream**. For each object, **size** calls are made to the **fputc** function, taking the values (in order) from an array of **unsigned char** exactly overlaying the object. The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

Returns

3 The **fwrite** function returns the number of elements successfully written, which will be less than **nmemb** only if a write error is encountered. If **size** or **nmemb** is zero, **fwrite** returns zero and the state of the stream remains unchanged.

7.19.9 File positioning functions

7.19.9.1 The fgetpos function

Synopsis

1
```
#include <stdio.h>
int fgetpos(FILE * restrict stream,
     fpos_t * restrict pos);
```

Description

2 The **fgetpos** function stores the current values of the parse state (if any) and file position indicator for the stream pointed to by **stream** in the object pointed to by **pos**. The values stored contain unspecified information usable by the **fsetpos** function for repositioning the stream to its position at the time of the call to the **fgetpos** function.

Returns

3 If successful, the **fgetpos** function returns zero; on failure, the **fgetpos** function returns nonzero and stores an implementation-defined positive value in **errno**.

Forward references: the **fsetpos** function (7.19.9.3).

7.19.9.2 The fseek function

Synopsis

1
```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```

Description

2 The **fseek** function sets the file position indicator for the stream pointed to by **stream**. If a read or write error occurs, the error indicator for the stream is set and **fseek** fails.

3 For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding **offset** to the position specified by **whence**. The specified position is the beginning of the file if **whence** is **SEEK_SET**, the current value of the file position indicator if **SEEK_CUR**, or end-of-file if **SEEK_END**. A binary stream need not meaningfully support **fseek** calls with a **whence** value of **SEEK_END**.

4 For a text stream, either **offset** shall be zero, or **offset** shall be a value returned by an earlier successful call to the **ftell** function on a stream associated with the same file and **whence** shall be **SEEK_SET**.

5 After determining the new position, a successful call to the **fseek** function undoes any effects of the **ungetc** function on the stream, clears the end-of-file indicator for the stream, and then establishes the new position. After a successful **fseek** call, the next operation on an update stream may be either input or output.

Returns

6 The **fseek** function returns nonzero only for a request that cannot be satisfied.

Forward references: the **ftell** function (7.19.9.4).

7.19.9.3 The fsetpos function

Synopsis

1
```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

2 The **fsetpos** function sets the **mbstate_t** object (if any) and file position indicator for the stream pointed to by **stream** according to the value of the object pointed to by **pos**, which shall be a value obtained from an earlier successful call to the **fgetpos** function on a stream associated with the same file. If a read or write error occurs, the error indicator for the stream is set and **fsetpos** fails.

3 A successful call to the **fsetpos** function undoes any effects of the **ungetc** function on the stream, clears the end-of-file indicator for the stream, and then establishes the new parse state and position. After a successful **fsetpos** call, the next operation on an update stream may be either input or output.

Returns

4 If successful, the **fsetpos** function returns zero; on failure, the **fsetpos** function returns nonzero and stores an implementation-defined positive value in **errno**.

7.19.9.4 The `ftell` function

Synopsis

1
```
#include <stdio.h>
long int ftell(FILE *stream);
```

Description

2 The **`ftell`** function obtains the current value of the file position indicator for the stream pointed to by **`stream`**. For a binary stream, the value is the number of characters from the beginning of the file. For a text stream, its file position indicator contains unspecified information, usable by the **`fseek`** function for returning the file position indicator for the stream to its position at the time of the **`ftell`** call; the difference between two such return values is not necessarily a meaningful measure of the number of characters written or read.

Returns

3 If successful, the **`ftell`** function returns the current value of the file position indicator for the stream. On failure, the **`ftell`** function returns −1L and stores an implementation-defined positive value in **`errno`**.

7.19.9.5 The `rewind` function

Synopsis

1
```
#include <stdio.h>
void rewind(FILE *stream);
```

Description

2 The **`rewind`** function sets the file position indicator for the stream pointed to by **`stream`** to the beginning of the file. It is equivalent to

```
(void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared.

Returns

3 The **`rewind`** function returns no value.

7.19.10 Error-handling functions

7.19.10.1 The `clearerr` function

Synopsis

1
```
#include <stdio.h>
void clearerr(FILE *stream);
```

Description

2 The **`clearerr`** function clears the end-of-file and error indicators for the stream pointed to by **`stream`**.

Returns

3 The **`clearerr`** function returns no value.

7.19.10.2 The `feof` function

Synopsis

1
```
#include <stdio.h>
int feof(FILE *stream);
```

Description

2 The **`feof`** function tests the end-of-file indicator for the stream pointed to by **`stream`**.

Returns

3 The **`feof`** function returns nonzero if and only if the end-of-file indicator is set for **`stream`**.

7.19.10.3 The `ferror` function

Synopsis

1
```
#include <stdio.h>
int ferror(FILE *stream);
```

Description

2 The **`ferror`** function tests the error indicator for the stream pointed to by **`stream`**.

Returns

3 The **`ferror`** function returns nonzero if and only if the error indicator is set for **`stream`**.

7.19.10.4 The `perror` function

Synopsis

1
```
#include <stdio.h>
void perror(const char *s);
```

Description

2 The **perror** function maps the error number in the integer expression **errno** to an error message. It writes a sequence of characters to the standard error stream thus: first (if **s** is not a null pointer and the character pointed to by **s** is not the null character), the string pointed to by **s** followed by a colon (**:**) and a space; then an appropriate error message string followed by a new-line character. The contents of the error message strings are the same as those returned by the **strerror** function with argument **errno**.

Returns

3 The **perror** function returns no value.

Forward references: the **strerror** function (7.21.6.2).