NAME

stdio – standard buffered input/output package

SYNOPSIS

#include <stdio.h>

extern **FILE** ∗**stdin;**

extern **FILE** ∗**stdout;**

extern **FILE** ∗**stderr;**

DESCRIPTION

The functions described in the entries of section 3S of this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros **getc( )** and **putc( )** handle characters quickly. The macros **getchar( )** and **putchar( )**, and the higher-level routines **fgetc( )**, **fgets( )**, **fprintf( )**, **fputc( )**, **fputs( )**, **fread( )**, **fscanf( )**, **fwrite( )**, **gets( )**, **getw( )**, **printf( )**, **puts( )**, **putw( )**, and **scanf( )** all use or act as if they use **getc( )** and **putc( )**; they can be freely intermixed.

A file with associated buffering is called a *stream* (see **intro**(3)) and is declared to be a pointer to a defined type **FILE**. **fopen( )** creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the **<stdio.h>** header and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

The following symbolic values in **<unistd.h>** define the file descriptors that will be associated with the C-language *stdin*, *stdout* and *stderr* when the application is started:

STDIN_FILENO	Standard input value	0	**stdin**
STDOUT_FILENO	Standard output value	1	**stdout**
STDERR_FILENO	Standard error value	2	**stderr**

The constant **NULL** designates a null pointer.

The integer-constant **EOF** is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

The integer constant **BUFSIZ** specifies the size of the buffers used by the particular implementation.

The integer constant **FILENAME_MAX** specifies the number of bytes needed to hold the longest pathname of a file allowed by the implementation. If the system does not impose a maximum limit, this value is the recommended size for a buffer intended to hold a file's pathname.

The integer constant **FOPEN_MAX** specifies the minimum number of files that the implementation guarantees can be open simultaneously. Note that no more than 255 files may be opened using **fopen( )**, and only file descriptors 0 through 255 can be used in a stream.

The functions and constants mentioned in the entries of section 3S of this manual are declared in that header and need no further declaration. The constants and the following ''functions'' are implemented as macros (redeclaration of these names is perilous): **getc( )**,

getchar( ), **putc( )**, **putchar( )**, **ferror( )**, **feof( )**, **clearerr( )**, and **fileno( )**. There are also function versions of **getc( )**, **getchar( )**, **putc( )**, **putchar( )**, **ferror**, **feof( )**, **clearerr( )**, and **fileno( )**.

Output streams, with the exception of the standard error stream **stderr**, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream **stderr** is by default unbuffered, but use of **freopen( )** (see **fopen**(3S)) will cause it to become buffered or line-buffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). **setbuf( )** or **setvbuf( )** (both described in **setbuf**(3S)) may be used to change the stream's buffering strategy.

Interactions of Other FILE-Type C Functions

A single open file description can be accessed both through streams and through file descriptors. Either a file descriptor or a stream will be called a *handle* on the open file description to which it refers; an open file description may have several handles.

Handles can be created or destroyed by user action without affecting the underlying open file description. Some of the ways to create them include **fcntl( ), dup( ), fdopen( ), fileno( ),** and **fork( )** (which duplicates existing ones into new processes). They can be destroyed by at least **fclose( ), close( ),** and the **exec** functions (which close some file descriptors and destroy streams).

A file descriptor that is never used in an operation, that could affect the file offset (for example **read**( ), **write**( ), or **lseek**( )) is not considered a handle in this discussion, but could give rise to one (as a consequence of **fdopen**( ), **dup**( ), or **fork**( ), for example). This exception does include the file descriptor underlying a stream, whether created with **fopen**( ) or **fdopen**( ), as long as it is not used directly by the application to affect the file offset. (The **read**( ) and **write**( ) functions implicitly affect the file offset; **lseek**( ) explicitly affects it.)

If two or more handles are used, and any one of them is a stream, their actions shall be coordinated as described below. If this is not done, the result is undefined.

A handle that is a stream is considered to be closed when either an **fclose**( ) or **freopen**( ) is executed on it (the result of **freopen**( ) is a new stream for this discussion, which cannot be a handle on the same open file description as its previous value) or when the process owning that stream terminates the **exit**( ) or **abort( ).** A file descriptor is closed by **close**( ), **_exit**( ), or by one of the **exec** functions when FD_CLOEXEC is set on that file descriptor.

For a handle to become the active handle, the actions below must be performed between the last other user of the first handle (the current active handle) and the first other user of the second handle (the future active handle). The second handle then becomes the active handle. All activity by the application affecting the file offset on the first handle shall be suspended until it again becomes the active handle. (If a stream function has as an underlying function that affects the file offset, the stream function will be considered to affect the file offset. The underlying functions are described below.)

The handles need not be in the same process for these rules to apply. Note that after a **fork**( ), two handles exist where one existed before. The application shall assure that, if both handles will ever be accessed, that they will both be in a state where the other could become the active handle first. The application shall prepare for a **fork**( ) exactly as if it were a change of active handle. (If the only action performed by one of the processes is one of the **exec** functions or **_exit**( ), the handle is never accessed in that process.)

(1) For the first handle, the first applicable condition below shall apply. After the actions required below are taken, the handle may be closed if it is still open.

 (a) If it is a file descriptor, no action is required.

 (b) If the only further action to be performed on any handle to this open file description is to close it, no action need be taken.

 (c) If it is a stream that is unbuffered, no action need be taken.

 (d) If it is a stream that is line-buffered and the last character written to the stream was a newline (that is, as if a **putc('\n')** was the most recent operation on that stream), no action need be taken.

 (e) If it is a stream that is open for writing or append (but not also open for reading), either an **fflush**( ) shall occur or the stream shall be closed.

 (f) If the stream is open for reading and it is at the end of the file (**feof**( ) is true), no action need be taken.

 (g) If the stream is open with a mode that allows reading and the underlying open file description refers to a device that is capable of seeking, either an **fflush**( ) shall occur or the stream shall be closed.

 (h) Otherwise, the result is undefined.

(2) For the second handle: if any previous active handle has called a function that explicitly changed the file offset, except as required above for the first handle, the application shall perform an **lseek**( ) or an **fseek**( ) (as appropriate to the type of the handle) to an appropriate location.

(3) If the active handle ceases to be accessible before the requirements on the first handle above have been met, the state of the open file description becomes undefined. This might occur, for example, during a **fork**( ) or an **_exit**( ).

(4) The **exec** functions shall be considered to make inaccessible all streams that are open at the time they are called, independent of what streams or file descriptors may be available to the new process image.

(5) Implementation shall assure that an application, even one consisting of several processes, shall yield correct results (no data is lost or duplicated when writing, all data is written in order, except as requested by seeks) when the rules above are followed, regardless of the sequence of handles used. If the rules above are not followed, the result is unspecified. When these rules are followed, it is implementation defined whether, and under what conditions, all input is seen exactly once.

Use of stdio in Multithreaded Applications

All the stdio functions are safe unless they have the _unlocked suffix. Each **file** pointer has its own lock to guarantee that only one thread can access it. In the case that output needs to be synchronized, the lock for the **FILE** pointer can be acquired before performing a series of stdio operations. For example:

```
FILE iop;
 .
 .
flockfile(iop);
fprintf(iop, "hello ");
fprintf(iop, "world0);
fputc(iop, 'a');
funlockfile(iop);
```

will print everything out together, blocking other threads that might want to write to the same file between fprintf's.

An unlocked interface is available in case performace is an issue. For example:

```
flockfile(iop);
while (!feof(iop)) {
        *c++ = getc_unlocked(iop);
}
funlockfile(iop);
```

RETURN VALUES

Invalid stream pointers usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

SEE ALSO

close(2), **lseek**(2), **open**(2), **pipe**(2), **read**(2), **write**(2), **ctermid**(3S), **cuserid**(3S), **fclose**(3S), **ferror**(3S), **fopen**(3S), **fread**(3S), **fseek**(3S), **flockfile**(3S), **getc**(3S), **gets**(3S), **popen**(3S), **printf**(3S), **putc**(3S), **puts**(3S), **scanf**(3S), **setbuf**(3S), **system**(3S), **tmpfile**(3S), **tmpnam**(3S), **ungetc**(3S)