13 Low-Level Input/Output

This chapter describes functions for performing low-level input/output operations on file descriptors. These functions include the primitives for the higher-level I/O functions described in Chapter 12 [Input/Output on Streams], page 247, as well as functions for performing low-level control operations for which there are no equivalents on streams.

Stream-level I/O is more flexible and usually more convenient; therefore, programmers generally use the descriptor-level functions only when necessary. These are some of the usual reasons:

- For reading binary files in large chunks.
- For reading an entire file into core before parsing it.
- To perform operations other than data transfer, which can only be done with a descriptor. (You can use `fileno` to get the descriptor corresponding to a stream.)
- To pass descriptors to a child process. (The child can create its own stream to use a descriptor that it inherits, but cannot inherit a stream directly.)

13.1 Opening and Closing Files

This section describes the primitives for opening and closing files using file descriptors. The `open` and `creat` functions are declared in the header file `fcntl.h`, while `close` is declared in `unistd.h`.

`int open` (*const char *`filename`, int `flags`*[, *mode_t `mode`*]) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `open` function creates and returns a new file descriptor for the file named by *filename*. Initially, the file position indicator for the file is at the beginning of the file. The argument *mode* (see Section 14.9.5 [The Mode Bits for Access Permission], page 407) is used only when a file is created, but it doesn't hurt to supply the argument in any case.

The *flags* argument controls how the file is to be opened. This is a bit mask; you create the value by the bitwise OR of the appropriate parameters (using the '`|`' operator in C). See Section 13.14 [File Status Flags], page 362, for the parameters available.

The normal return value from `open` is a non-negative integer file descriptor. In the case of an error, a value of −1 is returned instead. In addition to the usual file name errors (see Section 11.2.3 [File Name Errors], page 245), the following `errno` error conditions are defined for this function:

`EACCES` The file exists but is not readable/writable as requested by the *flags* argument, the file does not exist and the directory is unwritable so it cannot be created.

`EEXIST` Both `O_CREAT` and `O_EXCL` are set, and the named file already exists.

`EINTR` The `open` operation was interrupted by a signal. See Section 24.5 [Primitives Interrupted by Signals], page 685.

`EISDIR` The *flags* argument specified write access, and the file is a directory.

`EMFILE` The process has too many files open. The maximum number of file descriptors is controlled by the `RLIMIT_NOFILE` resource limit; see Section 22.2 [Limiting Resource Usage], page 630.

`ENFILE` The entire system, or perhaps the file system which contains the directory, cannot support any additional open files at the moment. (This problem cannot happen on GNU/Hurd systems.)

`ENOENT` The named file does not exist, and `O_CREAT` is not specified.

`ENOSPC` The directory or file system that would contain the new file cannot be extended, because there is no disk space left.

`ENXIO` `O_NONBLOCK` and `O_WRONLY` are both set in the *flags* argument, the file named by *filename* is a FIFO (see Chapter 15 [Pipes and FIFOs], page 422), and no process has the file open for reading.

`EROFS` The file resides on a read-only file system and any of `O_WRONLY`, `O_RDWR`, and `O_TRUNC` are set in the *flags* argument, or `O_CREAT` is set and the file does not already exist.

If on a 32 bit machine the sources are translated with `_FILE_OFFSET_BITS == 64` the function `open` returns a file descriptor opened in the large file mode which enables the file handling functions to use files up to 2^63 bytes in size and offset from -2^63 to 2^63. This happens transparently for the user since all of the lowlevel file handling functions are equally replaced.

This function is a cancellation point in multi-threaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores or whatever) at the time `open` is called. If the thread gets canceled these resources stay allocated until the program ends. To avoid this calls to `open` should be protected using cancellation handlers.

The `open` function is the underlying primitive for the `fopen` and `freopen` functions, that create streams.

`int open64` (*const char *`filename`, int `flags`*[, *mode_t `mode`*]) [Function]
Preliminary: | MT-Safe | AS-Safe | AC-Safe fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function is similar to `open`. It returns a file descriptor which can be used to access the file named by *filename*. The only difference is that on 32 bit systems the file is opened in the large file mode. I.e., file length and file offsets can exceed 31 bits.

When the sources are translated with `_FILE_OFFSET_BITS == 64` this function is actually available under the name `open`. I.e., the new, extended API using 64 bit file sizes and offsets transparently replaces the old API.

`int creat` (*const char *`filename`, mode_t `mode`*) [Obsolete function]
Preliminary: | MT-Safe | AS-Safe | AC-Safe fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function is obsolete. The call:

```
creat (filename, mode)
```

is equivalent to:

```
open (filename, O_WRONLY | O_CREAT | O_TRUNC, mode)
```

If on a 32 bit machine the sources are translated with `_FILE_OFFSET_BITS == 64` the function `creat` returns a file descriptor opened in the large file mode which enables the file handling functions to use files up to 2^63 in size and offset from -2^63 to 2^63. This happens transparently for the user since all of the lowlevel file handling functions are equally replaced.

`int creat64` (*const char *filename*, *mode_t mode*) [Obsolete function]
Preliminary: | MT-Safe | AS-Safe | AC-Safe fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function is similar to `creat`. It returns a file descriptor which can be used to access the file named by *filename*. The only the difference is that on 32 bit systems the file is opened in the large file mode. I.e., file length and file offsets can exceed 31 bits.

To use this file descriptor one must not use the normal operations but instead the counterparts named `*64`, e.g., `read64`.

When the sources are translated with `_FILE_OFFSET_BITS == 64` this function is actually available under the name `open`. I.e., the new, extended API using 64 bit file sizes and offsets transparently replaces the old API.

`int close` (*int filedes*) [Function]
Preliminary: | MT-Safe | AS-Safe | AC-Safe fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The function `close` closes the file descriptor *filedes*. Closing a file has the following consequences:

- The file descriptor is deallocated.
- Any record locks owned by the process on the file are unlocked.
- When all file descriptors associated with a pipe or FIFO have been closed, any unread data is discarded.

This function is a cancellation point in multi-threaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores or whatever) at the time `close` is called. If the thread gets canceled these resources stay allocated until the program ends. To avoid this, calls to `close` should be protected using cancellation handlers.

The normal return value from `close` is 0; a value of -1 is returned in case of failure. The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* argument is not a valid file descriptor.

`EINTR` The `close` call was interrupted by a signal. See Section 24.5 [Primitives Interrupted by Signals], page 685. Here is an example of how to handle `EINTR` properly:

```
TEMP_FAILURE_RETRY (close (desc));
```

`ENOSPC`
`EIO`
`EDQUOT` When the file is accessed by NFS, these errors from `write` can sometimes not be detected until `close`. See Section 13.2 [Input and Output Primitives], page 325, for details on their meaning.

Please note that there is *no* separate `close64` function. This is not necessary since this function does not determine nor depend on the mode of the file. The kernel which performs the `close` operation knows which mode the descriptor is used for and can handle this situation.

To close a stream, call `fclose` (see Section 12.4 [Closing Streams], page 252) instead of trying to close its underlying file descriptor with `close`. This flushes any buffered output and updates the stream object to indicate that it is closed.

13.2 Input and Output Primitives

This section describes the functions for performing primitive input and output operations on file descriptors: `read`, `write`, and `lseek`. These functions are declared in the header file `unistd.h`.

`ssize_t` [Data Type]
This data type is used to represent the sizes of blocks that can be read or written in a single operation. It is similar to `size_t`, but must be a signed type.

`ssize_t read` (*int* ***filedes***, *void* ****buffer***, *size_t* ***size***) [Function]
Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `read` function reads up to *size* bytes from the file with descriptor *filedes*, storing the results in the *buffer*. (This is not necessarily a character string, and no terminating null character is added.)

The return value is the number of bytes actually read. This might be less than *size*; for example, if there aren't that many bytes left in the file or if there aren't that many bytes immediately available. The exact behavior depends on what kind of file it is. Note that reading less than *size* bytes is not an error.

A value of zero indicates end-of-file (except if the value of the *size* argument is also zero). This is not considered an error. If you keep calling `read` while at end-of-file, it will keep returning zero and doing nothing else.

If `read` returns at least one character, there is no way you can tell whether end-of-file was reached. But if you did reach the end, the next read will return zero.

In case of an error, `read` returns -1. The following `errno` error conditions are defined for this function:

`EAGAIN` Normally, when no input is immediately available, `read` waits for some input. But if the `O_NONBLOCK` flag is set for the file (see Section 13.14 [File Status Flags], page 362), `read` returns immediately without reading any data, and reports this error.

Compatibility Note: Most versions of BSD Unix use a different error code for this: `EWOULDBLOCK`. In the GNU C Library, `EWOULDBLOCK` is an alias for `EAGAIN`, so it doesn't matter which name you use.

On some systems, reading a large amount of data from a character special file can also fail with `EAGAIN` if the kernel cannot find enough physical memory to lock down the user's pages. This is limited to devices that transfer with direct memory access into the user's memory, which means it does not include terminals, since they always use separate buffers inside the kernel. This problem never happens on GNU/Hurd systems.

Any condition that could result in `EAGAIN` can instead result in a successful `read` which returns fewer bytes than requested. Calling `read` again immediately would result in `EAGAIN`.

`EBADF` The *filedes* argument is not a valid file descriptor, or is not open for reading.

`EINTR` `read` was interrupted by a signal while it was waiting for input. See Section 24.5 [Primitives Interrupted by Signals], page 685. A signal will not necessary cause `read` to return `EINTR`; it may instead result in a successful `read` which returns fewer bytes than requested.

`EIO` For many devices, and for disk files, this error code indicates a hardware error.

`EIO` also occurs when a background process tries to read from the controlling terminal, and the normal action of stopping the process by sending it a `SIGTTIN` signal isn't working. This might happen if the signal is being blocked or ignored, or because the process group is orphaned. See Chapter 28 [Job Control], page 761, for more information about job control, and Chapter 24 [Signal Handling], page 659, for information about signals.

`EINVAL` In some systems, when reading from a character or block device, position and size offsets must be aligned to a particular block size. This error indicates that the offsets were not properly aligned.

Please note that there is no function named `read64`. This is not necessary since this function does not directly modify or handle the possibly wide file offset. Since the kernel handles this state internally, the `read` function can be used for all cases.

This function is a cancellation point in multi-threaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores or whatever) at the time `read` is called. If the thread gets canceled these resources stay allocated until the program ends. To avoid this, calls to `read` should be protected using cancellation handlers.

The `read` function is the underlying primitive for all of the functions that read from streams, such as `fgetc`.

`ssize_t pread` (*int* `filedes`, *void* *`buffer`, *size_t* `size`, *off_t* `offset`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `pread` function is similar to the `read` function. The first three arguments are identical, and the return values and error codes also correspond.

The difference is the fourth argument and its handling. The data block is not read from the current position of the file descriptor `filedes`. Instead the data is read from the file starting at position *offset*. The position of the file descriptor itself is not affected by the operation. The value is the same as before the call.

When the source file is compiled with `_FILE_OFFSET_BITS == 64` the `pread` function is in fact `pread64` and the type `off_t` has 64 bits, which makes it possible to handle files up to 2^63 bytes in length.

The return value of `pread` describes the number of bytes read. In the error case it returns -1 like `read` does and the error codes are also the same, with these additions:

`EINVAL` The value given for *offset* is negative and therefore illegal.

`ESPIPE` The file descriptor *filedes* is associate with a pipe or a FIFO and this device does not allow positioning of the file pointer.

The function is an extension defined in the Unix Single Specification version 2.

`ssize_t pread64` (*int* ***filedes***, *void* ***buffer***, *size_t* ***size***, *off64_t* ***offset***) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function is similar to the `pread` function. The difference is that the *offset* parameter is of type `off64_t` instead of `off_t` which makes it possible on 32 bit machines to address files larger than 2^31 bytes and up to 2^63 bytes. The file descriptor `filedes` must be opened using `open64` since otherwise the large offsets possible with `off64_t` will lead to errors with a descriptor in small file mode.

When the source file is compiled with `_FILE_OFFSET_BITS == 64` on a 32 bit machine this function is actually available under the name `pread` and so transparently replaces the 32 bit interface.

`ssize_t write` (*int* ***filedes***, *const void* ***buffer***, *size_t* ***size***) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `write` function writes up to *size* bytes from *buffer* to the file with descriptor *filedes*. The data in *buffer* is not necessarily a character string and a null character is output like any other character.

The return value is the number of bytes actually written. This may be *size*, but can always be smaller. Your program should always call `write` in a loop, iterating until all the data is written.

Once `write` returns, the data is enqueued to be written and can be read back right away, but it is not necessarily written out to permanent storage immediately. You can use `fsync` when you need to be sure your data has been permanently stored before continuing. (It is more efficient for the system to batch up consecutive writes and do them all at once when convenient. Normally they will always be written to disk within a minute or less.) Modern systems provide another function `fdatasync` which

guarantees integrity only for the file data and is therefore faster. You can use the `O_FSYNC` open mode to make `write` always store the data to disk before returning; see Section 13.14.3 [I/O Operating Modes], page 365.

In the case of an error, `write` returns −1. The following `errno` error conditions are defined for this function:

`EAGAIN` Normally, `write` blocks until the write operation is complete. But if the `O_NONBLOCK` flag is set for the file (see Section 13.11 [Control Operations on Files], page 358), it returns immediately without writing any data and reports this error. An example of a situation that might cause the process to block on output is writing to a terminal device that supports flow control, where output has been suspended by receipt of a STOP character.

Compatibility Note: Most versions of BSD Unix use a different error code for this: `EWOULDBLOCK`. In the GNU C Library, `EWOULDBLOCK` is an alias for `EAGAIN`, so it doesn't matter which name you use.

On some systems, writing a large amount of data from a character special file can also fail with `EAGAIN` if the kernel cannot find enough physical memory to lock down the user's pages. This is limited to devices that transfer with direct memory access into the user's memory, which means it does not include terminals, since they always use separate buffers inside the kernel. This problem does not arise on GNU/Hurd systems.

`EBADF` The *filedes* argument is not a valid file descriptor, or is not open for writing.

`EFBIG` The size of the file would become larger than the implementation can support.

`EINTR` The `write` operation was interrupted by a signal while it was blocked waiting for completion. A signal will not necessarily cause `write` to return `EINTR`; it may instead result in a successful `write` which writes fewer bytes than requested. See Section 24.5 [Primitives Interrupted by Signals], page 685.

`EIO` For many devices, and for disk files, this error code indicates a hardware error.

`ENOSPC` The device containing the file is full.

`EPIPE` This error is returned when you try to write to a pipe or FIFO that isn't open for reading by any process. When this happens, a `SIGPIPE` signal is also sent to the process; see Chapter 24 [Signal Handling], page 659.

`EINVAL` In some systems, when writing to a character or block device, position and size offsets must be aligned to a particular block size. This error indicates that the offsets were not properly aligned.

Unless you have arranged to prevent `EINTR` failures, you should check `errno` after each failing call to `write`, and if the error was `EINTR`, you should simply repeat the call. See Section 24.5 [Primitives Interrupted by Signals], page 685. The easy way to do this is with the macro `TEMP_FAILURE_RETRY`, as follows:

```
nbytes = TEMP_FAILURE_RETRY (write (desc, buffer, count));
```

Please note that there is no function named `write64`. This is not necessary since this function does not directly modify or handle the possibly wide file offset. Since the kernel handles this state internally the `write` function can be used for all cases.

This function is a cancellation point in multi-threaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores or whatever) at the time `write` is called. If the thread gets canceled these resources stay allocated until the program ends. To avoid this, calls to `write` should be protected using cancellation handlers.

The `write` function is the underlying primitive for all of the functions that write to streams, such as `fputc`.

`ssize_t pwrite` (*int* ***filedes***, *const void* ****buffer***, *size_t* ***size***, *off_t* ***offset***) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `pwrite` function is similar to the `write` function. The first three arguments are identical, and the return values and error codes also correspond.

The difference is the fourth argument and its handling. The data block is not written to the current position of the file descriptor `filedes`. Instead the data is written to the file starting at position *offset*. The position of the file descriptor itself is not affected by the operation. The value is the same as before the call.

When the source file is compiled with `_FILE_OFFSET_BITS == 64` the `pwrite` function is in fact `pwrite64` and the type `off_t` has 64 bits, which makes it possible to handle files up to 2^63 bytes in length.

The return value of `pwrite` describes the number of written bytes. In the error case it returns -1 like `write` does and the error codes are also the same, with these additions:

`EINVAL` The value given for *offset* is negative and therefore illegal.

`ESPIPE` The file descriptor *filedes* is associated with a pipe or a FIFO and this device does not allow positioning of the file pointer.

The function is an extension defined in the Unix Single Specification version 2.

`ssize_t pwrite64` (*int* ***filedes***, *const void* ****buffer***, *size_t* ***size***, *off64_t* ***offset***) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function is similar to the `pwrite` function. The difference is that the *offset* parameter is of type `off64_t` instead of `off_t` which makes it possible on 32 bit machines to address files larger than 2^31 bytes and up to 2^63 bytes. The file descriptor `filedes` must be opened using `open64` since otherwise the large offsets possible with `off64_t` will lead to errors with a descriptor in small file mode.

When the source file is compiled using `_FILE_OFFSET_BITS == 64` on a 32 bit machine this function is actually available under the name `pwrite` and so transparently replaces the 32 bit interface.

13.3 Setting the File Position of a Descriptor

Just as you can set the file position of a stream with `fseek`, you can set the file position of a descriptor with `lseek`. This specifies the position in the file for the next `read` or `write` operation. See Section 12.18 [File Positioning], page 304, for more information on the file position and what it means.

To read the current file position value from a descriptor, use `lseek (desc, 0, SEEK_CUR)`.

`off_t lseek` (*int* ***filedes***, *off_t* ***offset***, *int* ***whence***) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `lseek` function is used to change the file position of the file with descriptor *filedes*.

The *whence* argument specifies how the *offset* should be interpreted, in the same way as for the `fseek` function, and it must be one of the symbolic constants `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`.

`SEEK_SET` Specifies that *offset* is a count of characters from the beginning of the file.

`SEEK_CUR` Specifies that *offset* is a count of characters from the current file position. This count may be positive or negative.

`SEEK_END` Specifies that *offset* is a count of characters from the end of the file. A negative count specifies a position within the current extent of the file; a positive count specifies a position past the current end. If you set the position past the current end, and actually write data, you will extend the file with zeros up to that position.

The return value from `lseek` is normally the resulting file position, measured in bytes from the beginning of the file. You can use this feature together with `SEEK_CUR` to read the current file position.

If you want to append to the file, setting the file position to the current end of file with `SEEK_END` is not sufficient. Another process may write more data after you seek but before you write, extending the file so the position you write onto clobbers their data. Instead, use the `O_APPEND` operating mode; see Section 13.14.3 [I/O Operating Modes], page 365.

You can set the file position past the current end of the file. This does not by itself make the file longer; `lseek` never changes the file. But subsequent output at that position will extend the file. Characters between the previous end of file and the new position are filled with zeros. Extending the file in this way can create a "hole": the blocks of zeros are not actually allocated on disk, so the file takes up less space than it appears to; it is then called a "sparse file".

If the file position cannot be changed, or the operation is in some way invalid, `lseek` returns a value of -1. The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* is not a valid file descriptor.

`EINVAL` The *whence* argument value is not valid, or the resulting file offset is not valid. A file offset is invalid.

`ESPIPE` The *filedes* corresponds to an object that cannot be positioned, such as a pipe, FIFO or terminal device. (POSIX.1 specifies this error only for pipes and FIFOs, but on GNU systems, you always get `ESPIPE` if the object is not seekable.)

When the source file is compiled with `_FILE_OFFSET_BITS == 64` the `lseek` function is in fact `lseek64` and the type `off_t` has 64 bits which makes it possible to handle files up to 2^63 bytes in length.

This function is a cancellation point in multi-threaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores or whatever) at the time `lseek` is called. If the thread gets canceled these resources stay allocated until the program ends. To avoid this calls to `lseek` should be protected using cancellation handlers.

The `lseek` function is the underlying primitive for the `fseek`, `fseeko`, `ftell`, `ftello` and `rewind` functions, which operate on streams instead of file descriptors.

`off64_t lseek64` (*int* ***filedes***, *off64_t* ***offset***, *int* ***whence***) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function is similar to the `lseek` function. The difference is that the *offset* parameter is of type `off64_t` instead of `off_t` which makes it possible on 32 bit machines to address files larger than 2^31 bytes and up to 2^63 bytes. The file descriptor `filedes` must be opened using `open64` since otherwise the large offsets possible with `off64_t` will lead to errors with a descriptor in small file mode.

When the source file is compiled with `_FILE_OFFSET_BITS == 64` on a 32 bits machine this function is actually available under the name `lseek` and so transparently replaces the 32 bit interface.

You can have multiple descriptors for the same file if you open the file more than once, or if you duplicate a descriptor with `dup`. Descriptors that come from separate calls to `open` have independent file positions; using `lseek` on one descriptor has no effect on the other. For example,

```
{
  int d1, d2;
  char buf[4];
  d1 = open ("foo", O_RDONLY);
  d2 = open ("foo", O_RDONLY);
  lseek (d1, 1024, SEEK_SET);
  read (d2, buf, 4);
}
```

will read the first four characters of the file `foo`. (The error-checking code necessary for a real program has been omitted here for brevity.)

By contrast, descriptors made by duplication share a common file position with the original descriptor that was duplicated. Anything which alters the file position of one of the duplicates, including reading or writing data, affects all of them alike. Thus, for example,

```
{
  int d1, d2, d3;
  char buf1[4], buf2[4];
  d1 = open ("foo", O_RDONLY);
```

```
  d2 = dup (d1);
  d3 = dup (d2);
  lseek (d3, 1024, SEEK_SET);
  read (d1, buf1, 4);
  read (d2, buf2, 4);
}
```

will read four characters starting with the 1024'th character of `foo`, and then four more characters starting with the 1028'th character.

`off_t` [Data Type]

This is a signed integer type used to represent file sizes. In the GNU C Library, this type is no narrower than `int`.

If the source is compiled with `_FILE_OFFSET_BITS == 64` this type is transparently replaced by `off64_t`.

`off64_t` [Data Type]

This type is used similar to `off_t`. The difference is that even on 32 bit machines, where the `off_t` type would have 32 bits, `off64_t` has 64 bits and so is able to address files up to 2^63 bytes in length.

When compiling with `_FILE_OFFSET_BITS == 64` this type is available under the name `off_t`.

These aliases for the '`SEEK_...`' constants exist for the sake of compatibility with older BSD systems. They are defined in two different header files: `fcntl.h` and `sys/file.h`.

`L_SET` An alias for `SEEK_SET`.

`L_INCR` An alias for `SEEK_CUR`.

`L_XTND` An alias for `SEEK_END`.

13.4 Descriptors and Streams

Given an open file descriptor, you can create a stream for it with the `fdopen` function. You can get the underlying file descriptor for an existing stream with the `fileno` function. These functions are declared in the header file `stdio.h`.

`FILE * fdopen` (*int filedes*, *const char *opentype*) [Function]

Preliminary: | MT-Safe | AS-Unsafe heap lock | AC-Unsafe mem lock | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `fdopen` function returns a new stream for the file descriptor *filedes*.

The *opentype* argument is interpreted in the same way as for the `fopen` function (see Section 12.3 [Opening Streams], page 248), except that the '`b`' option is not permitted; this is because GNU systems make no distinction between text and binary files. Also, `"w"` and `"w+"` do not cause truncation of the file; these have an effect only when opening a file, and in this case the file has already been opened. You must make sure that the *opentype* argument matches the actual mode of the open file descriptor.

The return value is the new stream. If the stream cannot be created (for example, if the modes for the file indicated by the file descriptor do not permit the access specified by the *opentype* argument), a null pointer is returned instead.