

# write() — Write Data on a File or Socket

## Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

## Format

```
#define _POSIX_SOURCE
#include <unistd.h>

ssize_t write(int fs, const void *buf, size_t N);
```

## X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

ssize_t write(int fs, const void *buf, ssize_t N);
```

## Berkeley Sockets

```
#define _OE_SOCKETS
#include <unistd.h>

ssize_t write(int fs, const void *buf, ssize_t N);
```

## General Description

Writes *N* bytes from *buf* to the file or socket associated with *fs*. *N* should not be greater than `INT_MAX` (defined in the `limits.h` header file). If *N* is zero, `write()` simply returns 0 without attempting any other action.

If *fs* refers to a regular file or any other type of file on which a process can seek, `write()` begins writing at the file offset associated with *fs*. A successful `write()` increments the file offset by the number of bytes written. If the incremented file offset is greater than the previous length of the file, the length of the file is set to the new file offset.

If *fs* refers to a file on which a process cannot seek, `write()` begins writing at the current position. There is no file offset associated with such a file.

If `O_APPEND` (defined in the `fcntl.h` header file) is set for the file, `write()` sets the file offset to the end of the file before writing the output.

If there is not enough room to write the requested number of bytes (for example, because there is not enough room on the disk), `write()` outputs as many bytes as the remaining space can hold.

If `write()` is interrupted by a signal, the effect is one of the following:

- If `write()` has not written any data yet, it returns `-1` and sets `errno` to `EINTR`.
- If `write()` has successfully written some data, it returns the number of bytes it wrote before it was interrupted.

Write operations on pipes or FIFO special files are handled in the same way a write operation on a regular file, with the following exceptions:

- A pipe has no associated file offset, so every write appends to the end of the pipe.
- If  $N$  is less than or equal to `PIPE_BUF`, the output is not interleaved with data written by other processes that are writing to the same pipe. If  $N$  is greater than `PIPE_BUF` bytes, the output can be interleaved with other data (regardless of the setting of `O_NONBLOCK`, which is defined in the `fcntl.h` header file). A write to a pipe never returns with `errno` set to `EINTR` if it has transferred any data.
- If `O_NONBLOCK` (defined in the `fcntl.h` header file) is not set, `write()` may block process execution until normal completion.
- If `O_NONBLOCK` is set, `write()` does not block process execution. If  $N$  is less than or equal to `PIPE_BUF`, `write()` succeeds completely and returns the value of  $N$ , or else it writes nothing, sets `errno` to `EAGAIN`, and returns `-1`. If  $N$  is greater than `PIPE_BUF`, `write()` writes as many bytes as it can and returns this number as its result, or else it writes nothing, sets `errno` to `EAGAIN`, and returns `-1`.

With other files that support nonblocking writes and cannot accept data immediately, the effect is one of the following:

- If `O_NONBLOCK` is not set, `write()` blocks until the data can be written.
- If `O_NONBLOCK` is set, `write()` does not block the process. If some data can be written without blocking the process, `write()` writes what it can and returns the number of bytes written. Otherwise, it sets `errno` to `EAGAIN` and returns `-1`.

`write()` causes the signal `SIGTTOU` to be sent if all of these conditions are true:

- The process is attempting to write to its controlling terminal and `TOSTOP` is set as a terminal attribute.
- The process is running in a background process group and the `SIGTTOU` signal is not blocked or ignored.
- The process is not an orphan.

A successful `write()` updates the change and modification times for the file.

If *fs* refers to a socket, `write()` is equivalent to `send()` with no flags set.

## Behavior for Sockets

The `write()` function writes data from a buffer on a socket with descriptor *fs*. The socket must be a connected socket. This call writes up to *N* bytes of data.

### Parameter

	Description
<i>fs</i>	The file or socket descriptor.
<i>buf</i>	The pointer to the buffer holding the data to be written.
<i>N</i>	The length in bytes of the buffer pointed to by the <i>buf</i> parameter.

If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, `write()` blocks the caller until additional buffer space becomes available. If the socket is in nonblocking mode, `write()` returns -1 and sets the error code to `EWOULDBLOCK`. See [fcntl\(\) — Control Open File Descriptors](#) or [ioctl\(\) — Control Device](#) for a description of how to set the nonblocking mode.

When the socket is not ready to accept data and the process is trying to write data to the socket:

- Unless `FNDELAY` or `O_NONBLOCK` is set, `write()` blocks until the socket is ready to accept data.
- If `FNDELAY` is set, `write()` returns 0.
- If `O_NONBLOCK` is set, `write()` does not block the process. If some data can be written without blocking the process, `write()` writes what it can and returns the number of bytes written. Otherwise, it sets the error code to `EAGAIN` and returns -1.

For datagram sockets, this call sends the entire datagram, provided that the datagram fits into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application program wishes to send 1000 bytes, each call to this function can send 1 byte or 10 bytes or the entire 1000 bytes. Therefore, application programs using stream sockets should place this call in a loop, calling this function until all data has been sent.

## Special Behavior for C++ and Sockets

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

## Large Files for HFS

Note:

Large Files for HFS behavior is automatic for `AMODE 64` applications

Applications that are compiled with the option `LANGLVL(LONGLONG)` and also define the Feature Test Macro (FTM) `_LARGE_FILES` before any headers are included will enable this function to operate on HFS files that are larger than 2 gig-1 in size. File size and offset fields will be enlarged to 63 bits in width so any other function operating on this file will have to be enabled with the same FTM.

## Returned Value

If successful, `write()` returns the number of bytes actually written, less than or equal to  $N$ .

A value of 0 or greater indicates the number of bytes sent. However, this does not assure that data delivery was complete. A connection can be dropped by a peer socket and a `SIGPIPE` signal generated at a later time if data delivery is not complete.

If unsuccessful, `write()` returns -1 and sets `errno` to one of the following values:

### Error Code

Error Code	Description
<code>EAGAIN</code>	Resources temporarily unavailable. Subsequent requests may complete normally.
<code>EBADF</code>	$fs$ is not a valid file or socket descriptor.
<code>ECONNRESET</code>	A connection was forcibly closed by a peer.
<code>EDESTADDRREQ</code>	The socket is not connection-oriented and no peer address is set.
<code>EFAULT</code>	Using the <i>buf</i> and $N$ parameters would result in an attempt to access storage outside the caller's address space.
<code>EFBIG</code>	Writing to the output file would exceed the maximum file size supported by the implementation.
	An attempt was made to write a file that exceeds the system established maximum file size or the process's file size limit.
	The file is a regular file, $nbyte$ is greater than 0 and the starting position is greater than or equal to the offset maximum established in the open file description associated with fields.
<code>EINTR</code>	<code>write()</code> was interrupted by a signal before it had written any output.
<code>EINVAL</code>	The request is invalid or not supported. The <code>STREAM</code> or multiplexer referenced by $fs$ is linked (directly or indirectly) downstream from a multiplexer.
<code>EIO</code>	

The process is in a background process group and is attempting to write to its controlling terminal, but TOSTOP (defined in the `termios.h` header file) is set, the process is neither ignoring nor blocking SIGTTOU signals, and the process group of the process is orphaned. An I/O error occurred.

**EMSGSIZE**

The message was too big to be sent as a single datagram.

**ENOBUFS**

Buffer space is not available to send the message.

**ENOSPC**

There is no available space left on the output device.

**ENOTCONN**

The socket is not connected.

**ENXIO**

A hang-up occurred on the STREAM being written to.

**EPIPE**

`write()` is trying to write to a pipe that is not open for reading by any other process. This error also generates a SIGPIPE signal. For a connected stream socket the connection to the peer socket has been lost.

**ERANGE**

The transfer request size was outside the range supported by the STREAMS file associated with *fs*.

**EWOULDBLOCK**

|*socket* is in nonblocking mode and no data buffers |are available or the SO\_SNDTIMEO timeout value was reached before buffers |became available.

A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, `errno` is set to the value included in the error message.

Note:

z/OS UNIX System Services services do not supply any STREAMS devices or pseudodevices. It is impossible for `write()` to write any data on a STREAM. None of the STREAMS `errno`s will be visible to the invoker. See [open\(\) — Open a File](#)