read() — Read from a file or socket

Last Updated: 2021-06-25

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 read(int fs, void *buf, size_t N);
```

X/Open

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

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

Berkeley sockets

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

ssize_t read(int socket, void *buf, ssize_t N);
```

General description

From the file indicated by the file descriptor *fs*, the read() function reads *N* bytes of input into the memory area indicated by *buf*. A successful read() updates the access time for the file.

If *fs* refers to a regular file or any other type of file on which the process can seek, read() begins reading at the file offset associated with *fs*. If successful, read() changes the file offset by the number of bytes read. *N* should not be greater than INT_MAX (defined in the limits.h header file).

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

If *fs* refers to a socket, read() is equivalent to recv() with no flags set.

Parameter
Description
fs

The file or socket descriptor.

buf

The pointer to the buffer that receives the data.

N

The length in bytes of the buffer pointed to by the *buf* parameter.

Behavior for sockets: The read() call reads data on a socket with descriptor *fs* and stores it in a buffer. The read() all applies only to connected sockets. This call returns up to *N* bytes of data. If there are fewer bytes available than requested, the call returns the number currently available. If data is not available for the socket *fs*, and the socket is in blocking mode, the read() call blocks the caller until data arrives. If data is not available, and the socket is in nonblocking mode, read() returns a -1 and sets the error code to EWOULDBLOCK. See ioctl() – Control device or fcntl() – Control open file descriptors for a description of how to set nonblocking mode.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits into the specified buffer. Excess datagram data is discarded. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

Behavior for streams: A read() from a STREAMS file can read data in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the I_SRDOPT ioctl() request, and can be tested with the I_GRDOPT ioctl(). In byte-stream mode, read() retrieves data from the STREAM until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, read() retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached. If read() does not retrieve all the data in a message, the remaining data is left on the STREAM, and can be retrieved by the next read() call. Message-discard mode also retrieves data

until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the read() returns is discarded, and is not available for a subsequent read(), readv() or getmsg() call.

How read() handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, read() accepts data until it has read *N* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The read() function then returns the number of bytes read, and places the zero-byte message back on the STREAM to be retrieved by the next read(), readv() or getmsg(). In message-nondiscard mode or message-discard mode, a zero-byte message returns 0 and the message is removed from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message is removed from the STREAM and 0 is returned, regardless of the read mode.

A read() from a STREAMS file returns the data in the message at the front of the STREAM head read queue, regardless of the priority band of the message.

By default, STREAMs are in control-normal mode, in which a read() from a STREAMS file can only process messages that contain a data part but do not contain a control part. The read() fails if a message containing a control part is encountered at the STREAM head. This default action can be changed by placing the STREAM in either control-data mode or control-discard mode with the I_SRDOPT ioctl() command. In control-data mode, read() converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, read() discards message control parts but returns to the process any data part in the message.

In addition, read() and readv() will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of read() or readv() but reflects the prior error. If a hang-up occurs on the STREAM being read, read() continues to operate normally until the STREAM head read queue is empty. Thereafter, it returns 0.

Large file support for z/OS® UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX

files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, read() returns the number of bytes actually read and placed in *buf*. This number is less than or equal to *N*. It is less than *N* only if:

- read() reached the end of the file before reading the requested number of bytes.
- read() was interrupted by a signal.
- In POSIX C programs only, the file is a pipe, FIFO special file, or a character special file that has fewer than *N* bytes immediately available for reading.
- If the Physical File System does not support simple reads from directories, read() will return 0 if it is used for a directory. Users should use Opendir() and readdir() instead.

In POSIX C programs only, if read() is interrupted by a signal, the effect is one of the following:

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

If the starting position for the read operation is at the end of the file or beyond, read() returns 0.

In POSIX C programs, if read() attempts to read from an empty pipe or a FIFO special file, it has one of the following results:

- If no process has the pipe open for writing, read() returns 0 to indicate the end of the file.
- If some process has the pipe open for writing and O_NONBLOCK is set to 1, read() returns -1 and sets errno to EAGAIN.

- If some process has the pipe open for writing and O_NONBLOCK is set to 0, read() blocks (that is, does not return) until some data is written, or the pipe is closed by all other processes that have the pipe open for writing.

With other files that support nonblocking read operations (for example, character special files), a similar principle applies:

- If data is available, read() reads the data immediately.
- If no data is available and O_NONBLOCK is set to 1, read() returns -1 and sets errno to EAGAIN.
- If no data is available and O_NONBLOCK is set to 0, read() blocks until some data becomes available.

read() causes the signal SIGTTIN to be sent when all these conditions exist:

- The process is attempting to read from its controlling terminal.
- The process is running in a background process group.
- The SIGTTIN signal is not blocked or ignored.
- The process group of the process is not orphaned.

If read() is reading a regular file and encounters a part of the file that has not been written (but before the end of the file), read() places 0 bytes into *buf* in place of the unwritten bytes.

If the number of bytes of input that you want to read is 0, read() simply returns 0 without attempting any other action.

If the connection is broken on a stream socket, but data is available, then the read() function reads the data and gives no error. If the connection is broken on a stream socket, but no data is available, then the read() function returns 0 bytes as EOF.

Note: z/OS UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for read() to read any data from a STREAMS-based file indicated by *fs*. It will always return -1 with errno set to

> EBADF. EINVAL will never be set because there are no multiplexing STREAMS drivers. See open() – Open a file for more information.

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

Error Code	Description
EAGAIN	O_NONBLOCK is set to 1, but data was not available for reading.
EBADF	*fs* is not a valid file or socket descriptor.
ECONNRESET	A connection was forcibly closed by a peer.
EFAULT	Using the *buf* and *N* parameters would result in an attempt to access memory outside the caller's address space.
EINTR	read() was interrupted by a signal that was caught before any data was available.
EINVAL	*N* contains a value that is less than 0, or the request is invalid or not supported, or the STREAM or multiplexer referenced by *fs* is linked (directly or indirectly) downstream from a multiplexer.
EIO	The process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group of the process is orphaned. For sockets, an I/O error occurred.
ENOBUFS	Insufficient system resources are available to complete the call.
ENOTCONN	

A receive was attempted on a connection-oriented socket that is not connected.

EOVERFLOW

The file is a regular file and an attempt was made to read or write at or beyond the offset maximum associated with the file.

ETIMEDOUT

The connection timed out during connection establishment, or due to a transmission timeout on active connection.

EWOULDBLOCK

socket is in nonblocking mode and data is not available to read. or the SO_RCVTIMEO timeout value was been reached before data was available.

Example

CELEBR03

```
/* CELEBR03

   This example opens a file and reads input.

 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  int ret, fd;
  char buf[1024];

  system("ls -l / >| ls.output");

  if ((fd = open("ls.output", O_RDONLY)) < 0)
```

```
    perror("open() error");
  else {
    while ((ret = read(fd, buf, sizeof(buf)-1)) > 0) {
      buf[ret] = 0x00;
      printf("block read: \n<%s>\n", buf);
    }
    close(fd);
  }

  unlink("ls.output");
}
```

Output

```
block read:
<total 0
drwxr-xr-x   3 USER1    SYS1           0 Apr 16 07:59 bin
drwxr-xr-x   2 USER1    SYS1           0 Apr  6 10:20 dev
drwxr-xr-x   4 USER1    SYS1           0 Apr 16 07:59 etc
drwxr-xr-x   2 USER1    SYS1           0 Apr  6 10:15 lib
drwxrwxrwx   2 USER1    SYS1           0 Apr 16 07:55 tmp
drwxr-xr-x   2 USER1    SYS1           0 Apr  6 10:15 u
drwxr-xr-x   6 USER1    SYS1           0 Apr  6 10:15 usr
>
```

Related information

- limits.h – Standard values for limits on resources
- unistd.h – Implementation-specific functions
- close() – Close a file
- connect() – Connect a socket
- creat() – Create a new file or rewrite an existing one

- dup() – Duplicate an open file descriptor
- fcntl() – Control open file descriptors
- fread() – Read items
- getsockopt() – Get the options associated with a socket
- ioctl() – Control device
- lseek() – Change the offset of a file
- open() – Open a file
- pipe() – Create an unnamed pipe
- pread() – Read from a file or socket without file pointer change
- readv() – Read data on a file or socket and store in a set of buffers
- recv() – Receive data on a socket
- recvfrom() – Receive messages on a socket
- recvmsg() – Receive messages on a socket and store in an array of message headers
- select(), pselect() – Monitor activity on files or sockets and message queues
- selectex() – Monitor activity on files or sockets and message queues
- send() – Send data on a socket
- sendmsg() – Send messages on a socket
- sendto() – Send data on a socket
- setsockopt() – Set options associated with a socket
- socket() – Create a socket
- write() – Write data on a file or socket
- writev() – Write data on a file or socket from an array

Parent topic:

→ Library functions

Previous
random() – A better random-number generator

Next
readdir() – Read an entry from a directory