

open() — Open a File

Standards

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

Format

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

int open(const char *pathname, int options, ...);
```

General Description

Opens a file and returns a number called a *file descriptor*.

The *pathname* argument must be a hierarchical file system (HFS) file name. You can use this file descriptor to refer to the file in subsequent I/O operations, for example, read() or write(). Each file opened by a process gets a new file descriptor.

Restriction:

Using this function with FIFOs, POSIX terminals, and character special files requires z/OS XL C programs running POSIX(ON). See [Language Environment element](#) for more information.

The argument *pathname* is a string giving the name of the file you want to open. The integer *options* specifies options for the open operation by taking the bitwise inclusive-OR of symbols defined in the fcntl.h header file. The options indicate whether the file should be accessed for reading, writing, reading and writing, and so on.

An additional argument (...) is required if the O_CREAT option is specified in *options*. This argument may be called the *mode* and has the mode_t type. It specifies file permission bits to be used when a file is created. All the file permission bits are set to the bits of *mode*, except for those set in the file-mode creation mask of the process. Here is a list of symbols that can be used for a mode.

S_IRGRP

Read permission for the file's group.

S_IROTH

Read permission for users other than the file owner.

S_IRUSR
Read permission for the file owner.

S_IRWXG
Read, write, and search or execute permission for the file's group. S_IRWXG is the bitwise inclusive-OR of S_IRGRP, S_IWGRP, and S_IXGRP.

S_IRWXO
Read, write, and search or execute permission for users other than the file owner. S_IRWXO is the bitwise inclusive-OR of S_IROTH, S_IWOTH, and S_IXOTH.

S_IRWXU
Read, write, and search, or execute, for the file owner; S_IRWXG is the bitwise inclusive-OR of S_IRUSR, S_IWUSR, and S_IXUSR.

S_ISGID
Privilege to set group ID (GID) for execution. When this file is run through an exec function, the effective group ID of the process is set to the group ID of the file, so that the process has the same authority as the file owner rather than the authority of the actual invoker.

S_ISUID
Privilege to set the user ID (UID) for execution. When this file is run through an exec function, the effective user ID of the process is set to the owner of the file, so that the process has the same authority as the file owner rather than the authority of the actual invoker.

S_ISVTX
Indicates shared text. Keep loaded as an executable file in storage.

S_IWGRP
Write permission for the file's group.

S_IWOTH
Write permission for users other than the file owner.

S_IWUSR
Write permission for the file owner.

S_IXGRP
Search permission (for a directory) or execute permission (for a file) for the file's group.

S_IXOTH
Search permission for a directory, or execute permission for a file, for users other than the file owner.

S_IXUSR
Search permission (for a directory) or execute permission (for a file) for the file owner.

Most open operations position a *file offset* (an indicator showing where the next read or write will take place in the file) at the beginning of the file; however, there are options that can change this position. One of the following *must* be specified in the *options* argument of the open() operation:

O_RDONLY
Open for reading only

O_WRONLY

Open for writing only
O_RDWR
Open for both reading and writing

One or more of the following can also be specified in *options*:

O_APPEND
Positions the file offset at the end of the file before each write operation.

O_CREAT
Indicates that the call to open() has a *mode* argument.

If the file being opened already exists O_CREAT has no effect except when O_EXCL is also specified; see O_EXCL following.

If the file being opened does not exist it is created. The user ID is set to the effective ID of the process, and its group ID is set to the group ID of its directory. File permission bits are set according to *mode*.

If O_CREAT is specified and the file did not previously exist a successful open() sets the access time, change time, and modification time for the file. It also updates the change time and modification time fields in the parent directory.

O_EXCL

If both O_EXCL and O_CREAT are specified open() fails if the file already exists. If both O_EXCL and O_CREAT are specified and *pathname* names a symbolic link open() fails regardless of the contents of the symbolic link.

The check for the existence of the file and the creation of the file if it does not exist is atomic with respect to other threads executing open() naming the same filename in the same directory with O_EXCL and O_CREAT set.

O_NOCTTY

If *pathname* specifies a terminal open() does not make the terminal the controlling terminal of the process (and the session). If O_NOCTTY is not specified the terminal becomes the controlling terminal if the following conditions are true:

- The process is a session leader.
- There is no controlling terminal for the session.
- The terminal is not already a controlling terminal for another session.

O_NONBLOCK

Has different meanings depending on the situation.

- When you are opening a FIFO special file with O_RDONLY or O_WRONLY:

If `O_NONBLOCK` is specified a read-only `open()` returns immediately. A write-only `open()` returns with an error if no other process has the FIFO open for reading.

If `O_NONBLOCK` is not specified a read-only `open()` blocks until another process opens the FIFO for writing. A write-only `open()` blocks until another process opens the FIFO for reading.

- When you are opening a character special file that supports a nonblocking `open()`, `O_NONBLOCK` controls whether subsequent reads and writes can block.

`O_TRUNC`

If the file is successfully opened with `O_RDWR` or `O_WRONLY`, this will truncate the file to zero length if the file exists and is a regular file. The mode and owner of the file are unchanged. This option should not be used with `O_RDONLY`. `O_TRUNC` has no effect on FIFO special files or directories.

If `O_TRUNC` is specified and the file previously existed a successful `open()` updates the change time and modification time for the file.

`O_SYNC`

Force synchronous update. If this flag is 1 every `write()` operation on the file is written to permanent storage. That is, the file system buffers are forced to permanent storage. See `fsync()` also.

The program is assured that all data for the file has been written to permanent storage on return from a function which performs a synchronous update,

If *pathname* refers to a STREAM file, *oflag* may be constructed from `O_NONBLOCK` OR-ed with either `O_RDONLY`, `O_WRONLY` or `O_RDWR`. Other flag values are not applicable to STREAMS devices and have no effect on them. The value `O_NONBLOCK` affects the operation of STREAMS drivers and certain functions applied to file descriptors associated with STREAMS files. For STREAMS drivers, the implementation of `O_NONBLOCK` is device-specific.

Note:

z/OS UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for `open()` to return a valid STREAMS file descriptor.

The largest value that can be represented correctly in an object of type `off_t` is established as the offset maximum in the open file description.

Large Files for HFS

Note:

Large Files for HFS behavior is automatic for AMODE 64 applications

Applications that are compiled with the option `LANGVL(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, `open()` returns a file descriptor.

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

Error Code

Description

EACCES

Access is denied. Possible reasons include:

- The process does not have search permission on a component in *pathname*.
- The file exists, but the process does not have permission to open the file in the way specified by the flags.
- The file does not exist, and the process does not have write permission on the directory where the file is to be created.
- `O_TRUNC` was specified, but the process does not have write permission on the file.

EBUSY

The process attempted to open a file that is in use.

EEXIST

`O_CREAT` and `O_EXCL` were specified, and either the named file refers to a symbolic link, or the named file already exists.

EINTR

`open()` was interrupted by a signal.

EINVAL

The *options* parameter does not specify a valid combination of the `O_RDONLY`, `O_WRONLY` and `O_TRUNC` bits.

EIO

The *pathname* argument names a STREAMS file and a hang-up or error occurred during the `open()`.

EISDIR

pathname is a directory, and *options* specifies write or read/write access.

ELOOP

A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of *pathname* is greater than `POSIX_SYMLINK_MAX`.

EMFILE

The process has reached the maximum number of file descriptors it can have open.

ENAMETOOLONG

pathname is longer than **PATH_MAX** characters, or some component of *pathname* is longer than **NAME_MAX** characters while **_POSIX_NO_TRUNC** is in effect. For symbolic links, the length of the *pathname* string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined using `pathconf()`.

ENFILE

The system has reached the maximum number of file descriptors it can have open.

ENOENT

Typical causes:

- **O_CREAT** is not specified, and the named file does not exist.
- **O_CREAT** is specified, and either the prefix of *pathname* does not exist or the *pathname* argument is an empty string.

ENOMEM

The *pathname* argument names a STREAMS file and the system is unable to allocate resources.

ENOSPC

The directory or file system intended to hold a new file has insufficient space.

ENOSR

The *pathname* argument names a STREAMS-based file and the system is unable to allocate a STREAM.

ENOSYS

For master pseudoterminals, slave initialization did not complete.

ENOTDIR

A component of *pathname* is not a directory.

ENXIO

O_NONBLOCK and **O_WRONLY** were specified and the named file is a FIFO, but no process has the file open for reading. For a pseudoterminal, the requested minor number exceeds the maximum number supported by the installation.

EPERM

For slave pseudoterminals, permission to open is denied for one of these reasons:

- It is the first open of the slave after the master pseudoterminal was opened, and the user ID associated with the two opening processes is not the same.
- There was an internal error in the security system after the master pseudoterminal was opened.
- The attempt to open the slave used a different *pathname* than earlier opens used.

EROFS

pathname is on a read-only file system, and one or more of the options **O_WRONLY**, **O_RDWR**, **O_TRUNC**, or **O_CREAT** (if the file does not exist) was specified.

Example

The following opens an output file for appending:

```
int fd;  
fd = open("outfile",O_WRONLY | O_APPEND);
```

The following statement creates a new file with read/write/execute permissions for the creating user. If the file already exists, open() fails.

```
fd = open("newfile",O_WRONLY|O_CREAT|O_EXCL,S_IRWXU);
```