

fcntl() — Control Open File Descriptors

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 fcntl(int fildes, int action, ...);
```

X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int socket, int cmd, ...);
```

Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int socket, int cmd, ...);
```

General Description

Performs various actions on open file descriptors.

The argument *fildes* is a file descriptor for the file you want to manipulate. *action* is a symbol indicating the action you want to perform on *fildes*. These symbols are defined in the `<fcntl.h>` header file. If needed, “...” indicates a third argument. The type of the third argument depends on *action*, and some actions do not need an additional argument.

Behavior for Sockets

The operating characteristics of sockets can be controlled with the `fcntl()` call. The operations to be controlled are determined by *cmd*. The *arg* parameter is a variable with a meaning that depends on the value of the *cmd* parameter.

Parameter	Description
<i>socket</i>	The socket descriptor.
<i>cmd</i>	The command to perform.
<i>arg</i>	The data associated with <i>cmd</i> .

The *action* argument can be one of the following symbols:

F_CLOSEFD

Closes a range of file descriptors. A third `int` argument must be specified to indicate the upper limit for the range of the file descriptors to be closed, while *fildev* specifies the lower limit. If -1 is specified for the third argument, all file descriptors greater than or equal to the lower limit are closed.

F_DUPFD

Duplicates the file descriptor. A third `int` argument must be specified. `fcntl()` returns the lowest file descriptor greater than or equal to this third argument that is not already associated with an open file. This file descriptor refers to the same file as *fildev* and shares any locks. The flags `FD_CLOEXEC` and `FD_CLOFORK` are turned off in the new file descriptor, so that the file is kept open if an `exec` function is called.

Note:

If *fildev* is an XTI endpoint, there must be at least one available file descriptor greater than or equal to the third argument and less than 65536.

F_DUPFD2

Duplicates the file descriptor. A third `int` argument must be specified to indicate which file descriptor to use as the duplicate. This file descriptor is closed if already open and then used as the new file descriptor. The new file descriptor refers to the same file as *fildev* and shares any locks. The flags `FD_CLOEXEC` and `FD_CLOFORK` are turned off in the new file descriptor, so that the file is kept open if an `exec` function is called.

Note:

If *fildev* is an XTI endpoint, the third argument must not exceed the limit of 65535.

F_GETFD

Obtains the file descriptor flags for *fildev*. `fcntl()` returns these flags as its result. For a list of supported file descriptor flags, see [File Flags](#).

F_SETFD

Sets the file descriptor flags for *fildev*. You must specify a third `int` argument, giving the new file descriptor flag settings. `fcntl()` returns 0 if it successfully sets the flags.

F_GETFL

Obtains the file status flags and file access mode flags for *fildev*. `fcntl()` returns these flags as its result. For a list of supported file status and file access mode flags, see [File Flags](#).

Behavior for Sockets: This command gets the status flags of socket descriptor *socket*. With the `_OPEN_SYS` feature test macro you can query the `FNDELAY` flag. With the `_XOPEN_SOURCE_EXTENDED 1` feature test macro you can query the `O_NDELAY` flag. The `FNDELAY` and `O_NDELAY` flags mark *socket* as being in nonblocking mode. If data is not present on calls that can block, such as `read()`, `readv()`, and `recv()`, the call returns with -1, and the error code is set to `EWOULDBLOCK`.

F_SETFL

Sets the file status flags for *filides*. You must specify a third `int` argument, giving the new file descriptor flag settings. `fcntl()` does not change the file access mode, and file access bits in the third argument are ignored. `fcntl()` returns 0 if it successfully sets the flags.

Behavior for Sockets: This command sets the status flags of socket descriptor *socket*. With the `_OPEN_SYS` feature test macro you can set the `FNDELAY` flag. With the `_XOPEN_SOURCE_EXTENDED 1` feature test macro you can set the `O_NDELAY` flag.

F_GETLK

Obtains locking information for a file. See [File Locking](#)

F_SETLK

Sets or clears a file segment lock. See [File Locking](#)

F_SETLKW

Sets or clears a file segment lock; but if a shared or exclusive lock is blocked by other locks, `fcntl()` waits until the request can be satisfied. See [File Locking](#)

F_GETOWN

Behavior for Sockets: Obtains the PID for the *filides* and returns this value. The value returned will be either the process ID or the process group ID that is associated with the socket. If it is a positive integer, it specifies a process ID. If it is a negative integer (other than -1), it specifies a process group ID.

F_SETOWN

Behavior for Sockets: Sets either the process ID or the process group ID that is to receive either the `SIGIO` or `SIGURG` signals for the socket associated with *filides*. The `SIGURG` signal is generated as a result of receiving out-of-band data. Refer to `send()`, `sendto()`, `sendmsg()`, and `recv()`, `recvfrom()` and `recvmsg()` for more information on sending and receiving out-of-band data.

You must specify a third `int` argument, giving the PID requested. This value can be either a positive integer, specifying a process ID, or a negative integer (other than -1), specifying a process group ID. The difference between specifying a process ID or a process group ID is that in the first case only a single process will receive the signal, while in the second case all processes in the process group will receive the signal.

F_SETTAG

Sets the file tag for the file referred to by file descriptor *filides*.

The third argument *ftag* is the address of a populated `file_tag` structure.

If the *ftag* argument supplied to `fcntl(F_SETTAG)` does not have the `ft_deferred` bit set ON, `fcntl()` will immediately set the file's File Tag with the provided *ftag*'s `ft_ccsid` and `ft_txtflag` values.

If the *ftag* argument supplied to `fcntl(F_SETTAG)` has the `ft_deferred` bit set ON, `fcntl()` will not set the file's File Tag until first write to the file. The CCSID used to tag the file will be the current Program CCSID at the time of first write, regardless of the *ftag* `ft_ccsid` value, however the `ft_txtflag` value will be used.

If the `ft_ccsid` of the specified `file_tag` differs from the Program CCSID, automatic file conversion will occur, provided:

- The `ft_txtflag` is set to ON.
- The BPXPRMxx member `AUTOOCVT()` is ON or environment variable `_BPXK_AUTOOCVT` is ON.

If `AUTOOCVT(OFF)` and `_BPXK_AUTOOCVT=OFF`, the file will be tagged with the specified `file_tag`'s `ft_ccsid` and `ft_txtflag` values, but automatic conversion will not occur.

If the *ftag* argument supplied to `fcntl(F_SETTAG)` has the `ft_deferred` bit set ON, pipes and FIFOs are tagged from the write end with the Program CCSID of the first writer.

F_CONTROL_CVT

Controls or queries the conversion status of the open file referred to by file descriptor *fildev*. Conversion control is generally used to provide CCSID information for untagged files or untagged programs.

Character set conversion between a program and a file, pipe or other I/O stream can be enabled or changed with `F_CONTROL_CVT`. A pair of CCSID's is specified or defaulted, one for the program and one for the data. As the program reads and writes data, the system will convert from one CCSID to the other.

The third `f_cnvr` argument is the required address of an `f_cnvr` structure. This structure is defined in `<fcntl.h>` and includes the following members:

Table 22. Struct <code>f_cnvr</code> Element Descriptions		
Element	Data Type	Description
<code>pccsid</code>	<code>short</code>	The Program CCSID - This is output from query and input to setting conversion ON. A value of 0 on input indicates to use the previously set value or the current Program CCSID.
<code>fccsid</code>	<code>short</code>	The File CCSID - This is output from query and input to setting conversion ON. A value of 0 on input indicates to use the CCSID from the File Tag as stored in the file, specified on mount, or set by a prior

Table 22. Struct f_cnvrt Element Descriptions		
Element	Data Type	Description
		call.
cvtcmd	int	<p>Conversion Control Command. The following conversion controls are available:</p> <ul style="list-style-type: none"> • Query Conversion - Returns whether or not conversion is in effect and the Program and File CCSIDs being used. On input, <i>cvtcmd</i> is set to QUERYCVT, and on output, it is changed to either SETCVTON or SETCVTOFF to indicate that conversion is currently ON or OFF respectively. The current CCSIDs are also returned in their respective positions in the f_cnvrt structure. • Set Conversion OFF - Turns OFF any conversion that may be in effect. On input, <i>cvtcmd</i> is set to SETCVTOFF and the rest of the f_cnvrt is ignored. There is no output. A program can use this to override an automatic conversion that might be established by the environment within which it is invoked. If conversion is currently in effect, the CCSIDs being used will be remembered while conversion is turned OFF, so that the prior conversion may be resumed without the program having to remember what the prior CCSIDs were. • Set Conversion ON - Turns ON conversion and optionally specifies the CCSIDs to use in place of the Program or File CCSIDs that are currently in effect. A value of 0 for the Program CCSID indicates that the current Program CCSID be used. A value of 0 for the file CCSID indicates that no change should be made to the File CCSID. This does not affect the stored File Tag or the current Program CCSID. It only changes the values being used to control conversion on this data stream. • On input, <i>cvtcmd</i> is set to either SETCVTON to unconditionally turn on conversion or to SETAUTOCVTON to turn ON conversion only if AUTOCVT(YES) was specified in BPXPRMxx or _BPXK_AUTOCVT=ON.

The call fails if a conversion table is not installed for the resulting CCSID pair.

Warning:

Flipping the autoconversion mode off and on, or changing the CCSID values during the execution of a program in which file conversion and/or tagging takes place, or setting the CCSIDs to values that are not compatible with the program or file, can be quite unpredictable.

File Flags

There are several types of flags associated with each open file. Flags for a file are represented by symbols defined in the `<fcntl.h>` header file.

The following *file descriptor* flags can be associated with a file:

FD_CLOEXEC

If this flag is 1, the file descriptor is closed if the process executes one of the `exec` function calls. If it is 0, the file remains open.

FD_CLOFORK

If this flag is 1 when a fork occurs, the file descriptor will be closed for the child process. If it is 0, the file remains open for the child.

The following *file status* flags can be associated with a file:

O_APPEND

Append mode. If this flag is 1, every write operation on the file begins at the end of the file.

O_ASYNC

If this flag is 1, then asynchronous I/O will be used for the file.

O_NONBLOCK

No blocking. If this flag is 1, read and write operations on the file return with an error status if they cannot perform their I/O immediately. If this flag is 0, read and write operations on the file wait (or “block”) until the file is ready for I/O. For more details, see [read\(\) — Read From a File or Socket](#) and [write\(\) — Write Data on a File or Socket](#).

O_SYNC

Force synchronous update. If the 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\(\) — Write Changes to Direct-Access Storage](#).) If this flag is 0, update operations on the file will not be completed until the data has been written to permanent storage. On return from a function that performs a synchronous update, the program is assured that all data for the file has been written to permanent storage.

The following *file access mode* flags can be associated with a file:

O_RDONLY

The file is opened for reading only.

O_RDWR

The file is opened for reading and writing.

O_WRONLY

The file is opened for writing only.

Two masks can be used to extract flags:

O_ACCMODE

Extracts file access mode flags.

O_GETFL

Extracts file status flags and file access mode flags.

File Locking

A process can use `fcntl()` to lock out other processes from a part of a file, so that the process can read or write to that part of the file without interference from others. File locking can ensure data integrity when several processes have a file accessed concurrently. File locking can only be performed on file descriptors that refer to regular files. Locking is not permitted on file descriptors that refer to directories, FIFOs, pipes, character special files, or any other type of files.

A structure that has the type `struct flock` (defined in the `<fcntl.h>` header file) controls locking operations. This structure has the following members:

`short l_type`

Indicates the type of lock, using one of the following symbols (defined in the `<fcntl.h>` header file):

`F_RDLCK`

Indicates a *read lock*, also called a *shared lock*. The process can read the locked part of the file, and other processes cannot obtain write locks for that part of the file in the meantime. More than one process can have a read lock on the same part of a file simultaneously.

To establish a read lock, a process must have the file accessed for reading.

`F_WRLCK`

Indicates a *write lock*, also called an *exclusive lock*. The process can write on the locked part of the file, and no other process can establish a read lock or write lock on that same part or on an overlapping part of the file. A process cannot put a write lock on part of a file if there is already a read lock on an overlapping part of the file. To establish a write lock, a process must have accessed the file for writing.

`F_UNLCK`

Unlocks a lock that was set previously. An unlock (`F_UNLCK`) request in which `l_len` is non-zero and the offset of the last byte of the requested segment is the maximum value for an object of type `off_t`, when the process has an existing lock in which `l_len` is 0 and which includes the last byte of the requested segment, is treated as a request to unlock from the start of the requested segment with an `l_len` equal to 0. Otherwise, an unlock (`F_UNLCK`) request attempts to unlock only the requested segment.

`short l_whence`

One of three symbols used to determine the part of the file that is affected by this lock. These symbols are defined in the `<unistd.h>` header file and are the same as symbols used by `lseek()`:

`SEEK_CUR`

The current file offset in the file

`SEEK_END`

The end of the file

`SEEK_SET`

The start of the file.

`off_t l_start`
 Gives the byte offset used to identify the part of the file that is affected by this lock. The part of the file affected by the lock begins at this offset from the location given by `l_whence`. For example, if `l_whence` is `SEEK_SET` and `l_start` is 10, the locked part of the file begins at an offset of 10 bytes from the beginning of the file.

`off_t l_len`
 Gives the size of the locked part of the file in bytes. If `l_len` is 0, the locked part of the file begins at the position specified by `l_whence` and `l_start`, and extends to the end of the file. If `l_len` is positive, the area affected starts at `l_start` and end at `l_start + l_len - 1`. If `l_len` is negative, the area affected starts at `l_start + l_len` and end at `l_start - 1`. Locks may start and extend beyond the current end of a file, but cannot extend before the beginning of the file. A lock can be set to extend to the largest possible value of the file offset for that file by setting `l_len` to 0. If such a lock also has `l_start` set to 0 and `l_whence` is set to `SEEK_SET`, the whole file is locked.

`pid_t l_pid`
 Specifies the process ID of the process that holds the lock. This is an output field used only with `F_GETLK` actions.

You can set locks by specifying `F_SETLK` as the *action* argument for `fcntl()`. Such a function call requires a third argument pointing to a `struct flock` structure, as in this example:

```
struct flock lock_it;
lock_it.l_type = F_RDLCK;
lock_it.l_whence = SEEK_SET;
lock_it.l_start = 0;
lock_it.l_len = 100;
fcntl(fildes, F_SETLK, &lock_it);
```

This example sets up an `flock` structure describing a read lock on the first 100 bytes of a file, and then calls `fcntl()` to establish the lock. You can unlock this lock by setting `l_type` to `F_UNLCK` and making the same call. If an `F_SETLK` operation cannot set a lock, it returns immediately with an error saying that the lock cannot be set.

The `F_SETLKW` operation is similar to `F_SETLK`, except that it waits until the lock can be set. For example, if you want to establish an exclusive lock and some other process already has a lock established on an overlapping part of the file, `fcntl()` waits until the other process has removed its lock. If `fcntl()` is waiting in an `F_SETLKW` operation when a signal is received, `fcntl()` is interrupted. After handling the signal, `fcntl()` returns -1 and sets `errno` to `EINTR`.

`F_SETLKW` operations can encounter *deadlocks* when process A is waiting for process B to unlock a region, and B is waiting for A to unlock a different region. If the system detects that an `F_SETLKW` might cause a deadlock, `fcntl()` fails with `errno` set to `EDEADLK`.

A process can determine locking information about a file by using `F_GETLK` as the *action* argument for `fcntl()`. In this case, the call to `fcntl()` should specify a third argument pointing to an `flock` structure. The structure should describe the lock operation you want. When `fcntl()` returns, the structure indicated by the `flock` pointer is changed to show the first lock that would prevent

the proposed lock operation from taking place. The returned structure shows the type of lock that is set, the part of the file that is locked, and the process ID of the process that holds the lock. In the returned structure:

- `l_whence` is always `SEEK_SET`.
- `l_start` gives the offset of the locked portion from the beginning of the file.
- `l_len` is the length of the locked portion.

If there are no locks that prevent the proposed lock operation, the returned structure has `F_UNLCK` in `l_type`, and is otherwise unchanged.

A process can have several locks on a file simultaneously but only one type of lock set on a given byte. Therefore, if a process puts a new lock on part of a file that it had locked previously, the process has only one lock on that part of the file: the type of the lock is the one specified in the most recent locking operation.

All of a process's locks on a file are removed when the process closes any file descriptor that refers to the locked file. Locks are not inherited by child processes created with `fork()`.

All locks are advisory only. Processes can use locks to inform each other that they want to protect parts of a file, but locks do not prevent I/O on the locked parts. If a process has appropriate permissions on a file, it can perform whatever I/O it chooses, regardless of what locks are set. Therefore, file locking is only a convention, and it works only when all processes respect the convention.

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.

Usage of the `F_GETFL` command will return the setting of `O_LARGEFILE` status flag when `fcntl()` has been enabled to operate on large files

Returned Value

If successful, the value `fcntl()` returns will depend on the *action* that was specified.

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

Error Code

Description

EAGAIN

The process tried to set a lock with `F_SETLK`, but the lock is in conflict with a lock already set by some other process on an overlapping part of the file.

EBADF

fdes is not a valid open file descriptor; or the process tried to set a read lock on a file descriptor open for writing only; or the process tried to set a write lock on a file descriptor open for reading only; or the *socket* parameter is not a valid socket descriptor.

In an `F_DUPFD2` operation, the third argument is negative, or greater than or equal to **OPEN_MAX**, which is the highest file descriptor value allowed for the process.

EDEADLK

The system detected the potential for deadlock in a `F_SETLKW` operation.

EINTR

`fcntl()` was interrupted by a signal during a `F_SETLKW` operation.

EINVAL

In an `F_DUPFD` operation, the third argument is negative or greater than or equal to **OPEN_MAX**, the highest file descriptor value allowed for the process. The **OPEN_MAX** value can be determined using `pathconf()`.

In a locking operation, *fdes* refers to a file with a type that does not support locking, or the `struct flock` pointed to by the third argument has an incorrect form.

If an `F_CLOSF` operation, the third argument, which specifies the upper limit, is less than *filedes* but is not equal to -1.

Behavior for Sockets: The *arg* parameter is not a valid flag, or the *cmd* parameter is not a valid command.

EMFILE

In an `F_DUPFD` operation, the process has already reached its maximum number of file descriptors, or there are no available file descriptors greater than the specified third argument.

ENOLCK

In an `F_SETLK` or `F_SETLKW` operation, the specified file has already reached the maximum number of locked regions allowed by the system.

EOVERFLOW

One of the values to be returned cannot be represented correctly.

The *cmd* argument is `F_GETLK`, `F_SETLK` or `F_SETLKW` and the smallest or, if *l_len* is nonzero, the largest offset of any byte in the requested segment cannot be represented correctly in an object of type `off_t`.

EPERM

The operation was `F_CLOSF`, but all the requested file descriptors were not closed.

