

UNIX-Type APIs (V5R2)

Integrated File System (IFS) APIs

Volume 1 -- access() through pwrite64()

Table of Contents

The PDF for the Integrated File System (IFS) APIs is divided into two volumes. Volume 1 contains the APIs from access() through pwrite64(); Volume 2 contains the APIs QlgAccess() through writev() and the IFS exit programs. Both volumes contain information on time stamp updates, the Header Files for UNIX-Type Functions, and Errno Values for UNIX-Type Functions.

[Integrated File System APIs](#)

- [access\(\)](#) (Determine file accessibility)
- [accessx\(\)](#) (Determine File Accessibility for a Class of Users)⌘
- [chdir\(\)](#) (Change current directory)
- [chmod\(\)](#) (Change file authorizations)
- [chown\(\)](#) (Change owner and group of file)
- [close\(\)](#) (Close file descriptor)
- [closedir\(\)](#) (Close directory)
- [creat\(\)](#) (Create new file or rewrite existing file)
- [creat64\(\)](#) (Create new file or rewrite existing file (large file enabled))
- [DosSetFileLocks\(\)](#) (Lock and unlock a range of an open file)
- [DosSetFileLocks64\(\)](#) (Lock and unlock a range of an open file (large file enabled))
- [DosSetRelMaxFH\(\)](#) (Change maximum number of file descriptors)
- [dup\(\)](#) (Duplicate open file descriptor)
- [dup2\(\)](#) (Duplicate open file descriptor to another descriptor)
- [faccessx\(\)](#) (Determine File Accessibility for a Class of Users)⌘
- [fchdir\(\)](#) (Change Current Directory by Descriptor)⌘
- [fchmod\(\)](#) (Change file authorizations by descriptor)
- [fchown\(\)](#) (Change owner and group of file by descriptor)
- [fcntl\(\)](#) (Perform file control command)
- [fpathconf\(\)](#) (Get configurable path name variables by descriptor)
- [fstat\(\)](#) (Get file information by descriptor)
- [fstat64\(\)](#) (Get file information by descriptor (large file enabled))
- [fstatvfs\(\)](#) (Get file system information by descriptor)
- [fstatvfs64\(\)](#) (Get file system information by descriptor (64-bit enabled))

- [fsync\(\)](#) (Synchronize changes to file)
- [ftruncate\(\)](#) (Truncate file)
- [ftruncate64\(\)](#) (Truncate file (large file enabled))
- [getcwd\(\)](#) (Get current directory)
- [getegid\(\)](#) (Get effective group ID)
- [geteuid\(\)](#) (Get effective user ID)
- [getgid\(\)](#) (Get real group ID)
- [getgrgid\(\)](#) (Get group information using group ID)
- [getgrgid_r\(\)](#) (Get group information using group ID)
- [getgrgid_r_ts64\(\)](#) (Get group information using group ID)
- [getgrnam\(\)](#) (Get group information using group name)
- [getgrnam_r\(\)](#) (Get group information using group name)
- [getgrnam_r_ts64\(\)](#) (Get group information using group name)
- [getgroups\(\)](#) (Get group IDs)
- [getpwnam\(\)](#) (Get user information for user name)
- [getpwnam_r\(\)](#) (Get User Information for User Name)
- [getpwnam_r_ts64\(\)](#) (Get user information for user name)
- [getpwuid\(\)](#) (Get user information for user ID)
- [getpwuid_r\(\)](#) (Get User Information for User ID)
- [getpwuid_r_ts64\(\)](#) (Get user information for user ID)
- [getuid\(\)](#) (Get real user ID)
- [ioctl\(\)](#) (Perform file I/O control request)
- [lchown\(\)](#) (Change owner and group of symbolic link)
- [link\(\)](#) (Create link to file)
- [lseek\(\)](#) (Set file read/write offset)
- [lseek64\(\)](#) (Set file read/write offset (large file enabled))
- [lstat\(\)](#) (Get file or link information)
- [lstat64\(\)](#) (Get file or link information (large file enabled))
- [mkdir\(\)](#) (Make directory)
- [mkfifo\(\)](#) (Make FIFO special file)
- [mmap\(\)](#) (Memory map a file)
- [mmap64\(\)](#) (Memory map a stream file (large file enabled))
- [mprotect\(\)](#) (Change access protection for memory mapping)
- [msync\(\)](#) (Synchronize modified data with mapped file)
- [munmap\(\)](#) (Remove memory mapping)
- [open\(\)](#) (Open file)

- [open64\(\)](#) (Open file (large file enabled))
- [opendir\(\)](#) (Open directory)
- [pathconf\(\)](#) (Get configurable path name variables)
- [pipe\(\)](#) (Create interprocess channel)
- [pread\(\)](#) (Read from Descriptor with Offset)❏
- [pread64\(\)](#) (Read from Descriptor with Offset (large file enabled))❏
- [pwrite\(\)](#) (Write to Descriptor with Offset)❏
- [pwrite64\(\)](#) (Write to Descriptor with Offset (large file enabled))❏

[Integrated File System APIs--Time Stamp Updates](#)

[Header Files for UNIX-Type Functions](#)


[Errno Values for UNIX-Type Functions](#)

Integrated File System APIs

access() through pwrite64()

The integrated file system APIs can perform operations on directories, files, and related objects in the file systems accessed through the integrated file system interface.

The integrated file system APIs (access() through pwrite64()) are:

- [access\(\)](#) (Determine file accessibility) determines whether a file can be accessed in a particular manner.
- [accessx\(\)](#) (Determine File Accessibility for a Class of Users) determines whether a file can be accessed by a specified class of users in a particular manner. 
- [chdir\(\)](#) (Change current directory) makes the directory named by path the new current directory.
- [chmod\(\)](#) (Change file authorizations) changes the mode of the file or directory specified in path.
- [chown\(\)](#) (Change owner and group of file) changes the owner and group of a file.
- [close\(\)](#) (Close file descriptor) closes a descriptor, fildes.
- [closedir\(\)](#) (Close directory) closes the directory stream indicated by dirp.
- [creat\(\)](#) (Create new file or rewrite existing file) creates a new file or rewrites an existing file so that it is truncated to zero length.
- [creat64\(\)](#) (Create new file or rewrite existing file (large file enabled)) creates a new file or rewrites an existing file so that it is truncated to zero length.
- [DosSetFileLocks\(\)](#) (Lock and unlock a range of an open file) locks and unlocks a range of an open file.
- [DosSetFileLocks64\(\)](#) (Lock and unlock a range of an open file (large file enabled)) locks and unlocks a range of an open file.
- [DosSetRelMaxFH\(\)](#) (Change maximum number of file descriptors) requests that the system change the maximum number of file descriptors for the calling process (job).
- [dup\(\)](#) (Duplicate open file descriptor) returns a new open file descriptor.
- [dup2\(\)](#) (Duplicate open file descriptor to another descriptor) returns a descriptor with the value fildes2.
- [faccessx\(\)](#) (Determine File Accessibility for a Class of Users) determines whether a file can be accessed by a specified class of users in a particular manner. 
- [fchdir\(\)](#) (Change Current Directory by Descriptor) makes the directory named by fildes the new current directory. 
- [fchmod\(\)](#) (Change file authorizations by descriptor) sets the file permission bits of the open file identified by fildes, its file descriptor.
- [fchown\(\)](#) (Change owner and group of file by descriptor) changes the owner and group of a file.
- [fcntl\(\)](#) (Perform file control command) performs various actions on open descriptors.
- [fpathconf\(\)](#) (Get configurable path name variables by descriptor) determines the value of a configuration variable (name) associated with a particular file descriptor (file_descriptor).
- [fstat\(\)](#) (Get file information by descriptor) gets status information about the file specified by the open file descriptor file_descriptor and stores the information in the area of memory indicated by

the buf argument.

- [fstat64\(\)](#) (Get file information by descriptor (large file enabled)) gets status information about the file specified by the open file descriptor `file_descriptor` and stores the information in the area of memory indicated by the buf argument.
- [fstatvfs\(\)](#) (Get file system information by descriptor) gets status information about the file system that contains the file referenced by the open file descriptor `file_descriptor`.
- [fstatvfs64\(\)](#) (Get file system information by descriptor (64-bit enabled)) gets status information about the file system that contains the file referred to by the open file descriptor `file_descriptor`.
- [fsync\(\)](#) (Synchronize changes to file) transfers all data for the file indicated by the open file descriptor `file_descriptor` to the storage device associated with `file_descriptor`.
- [ftruncate\(\)](#) (Truncate file) truncates the file indicated by the open file descriptor `file_descriptor` to the indicated length.
- [ftruncate64\(\)](#) (Truncate file (large file enabled)) truncates the file indicated by the open file descriptor `file_descriptor` to the indicated length.
- [getcwd\(\)](#) (Get current directory) determines the absolute path name of the current directory and stores it in buf.
- [getegid\(\)](#) (Get effective group ID) returns the effective group ID (gid) of the calling thread.
- [geteuid\(\)](#) (Get effective user ID) returns the effective user ID (uid) of the calling thread.
- [getgid\(\)](#) (Get real group ID) returns the real group ID (gid) of the calling thread.
- [getgrgid\(\)](#) (Get group information using group ID) returns a pointer to an object of type struct group containing an entry from the user database with a matching gid.
- [getgrgid_r\(\)](#) (Get group information using group ID) updates the group structure pointed to by `grp` and stores a pointer to that structure in the location pointed to by `result`.
- [getgrgid_r_ts64\(\)](#) (Get group information using group ID) updates the group structure pointed to by `grp` and stores a pointer to that structure in the location pointed to by `result`.
- [getgrnam\(\)](#) (Get group information using group name) returns a pointer to an object of type struct group containing an entry from the user database with a matching name.
- [getgrnam_r\(\)](#) (Get group information using group name) updates the group structure pointed to by `grp` and stores a pointer to that structure in the location pointed to by `result`.
- [getgrnam_r_ts64\(\)](#) (Get group information using group name) updates the group structure pointed to by `grp` and stores a pointer to that structure in the location pointed to by `result`.
- [getgroups\(\)](#) (Get group IDs) returns the number of primary and supplementary group IDs associated with the calling thread without modifying the array pointed to by the `grouplist` argument.
- [getpwnam\(\)](#) (Get user information for user name) returns a pointer to an object of type struct passwd containing an entry from the user database with a matching name.
- [getpwnam_r\(\)](#) (Get User Information for User Name) updates the `passwd` structure pointed to by `pwd` and stores a pointer to that structure in the location pointed to by `result`.
- [getpwnam_r_ts64\(\)](#) (Get user information for user name) updates the `passwd` structure pointed to by `pwd` and stores a pointer to that structure in the location pointed to by `result`.
- [getpwuid\(\)](#) (Get user information for user ID) returns a pointer to an object of type struct passwd containing an entry from the user database with a matching uid.
- [getpwuid_r\(\)](#) (Get User Information for User ID) updates the `passwd` structure pointed to by `pwd` and stores a pointer to that structure in the location pointed to by `result`.

- [getpwuid_r_ts64\(\)](#) (Get user information for user ID) updates the passwd structure pointed to by pwd and stores a pointer to that structure in the location pointed to by result.
- [getuid\(\)](#) (Get real user ID) returns the real user ID (uid) of the calling thread.
- [ioctl\(\)](#) (Perform file I/O control request) performs control functions (requests) on a file descriptor.
- [lchown\(\)](#) (Change owner and group of symbolic link) changes the owner and group of a file. If the named file is a symbolic link, lchown() changes the owner or group of the link itself rather than the object to which the link points.
- [link\(\)](#) (Create link to file) provides an alternative path name for the existing file, so that the file can be accessed by either the existing name or the new name.
- [lseek\(\)](#) (Set file read/write offset) changes the current file offset to a new position in the file.
- [lseek64\(\)](#) (Set file read/write offset (large file enabled)) changes the current file offset to a new position in the file.
- [lstat\(\)](#) (Get file or link information) gets status information about a specified file and places it in the area of memory pointed to by buf.
- [lstat64\(\)](#) (Get file or link information (large file enabled)) gets status information about a specified file and places it in the area of memory pointed to by buf.
- [mkdir\(\)](#) (Make directory) creates a new, empty directory whose name is defined by path.
- [mkfifo\(\)](#) (Make FIFO special file) creates a new FIFO special file (FIFO) whose name is defined by path.
- [mmap\(\)](#) (Memory map a file) establishes a mapping between a process' address space and a stream file.
- [mmap64\(\)](#) (Memory map a stream file (large file enabled)) is used to establish a memory mapping of a file.
- [mprotect\(\)](#) (Change access protection for memory mapping) is used to change the access protection of a memory mapping to that specified by protection.
- [msync\(\)](#) (Synchronize modified data with mapped file) can be used to write modified data from a shared mapping (created using the mmap() function) to non-volatile storage or invalidate privately mapped pages.
- [munmap\(\)](#) (Remove memory mapping) removes addressability to a range of memory mapped pages of a process's address space.
- [open\(\)](#) (Open file) opens a file and returns a number called a file descriptor.
- [open64\(\)](#) (Open file (large file enabled)) opens a file and returns a number called a file descriptor.
- [opendir\(\)](#) (Open directory) opens a directory so that it can be read with the readdir() function.
- [pathconf\(\)](#) (Get configurable path name variables) lets an application determine the value of a configuration variable (name) associated with a particular file or directory (path).
- [pipe\(\)](#) (Create interprocess channel) creates a data pipe and places two file descriptors, one each into the arguments fildes[0] and fildes[1], that refer to the open file descriptions for the read and write ends of the pipe, respectively.
- [pread\(\)](#) (Read from Descriptor with Offset) reads *nbyte* bytes of input into the memory area indicated by *buf*.↵
- [pread64\(\)](#) (Read from Descriptor with Offset (large file enabled)) reads *nbyte* bytes of input into the memory area indicated by *buf*.↵
- [pwrite\(\)](#) (Write to Descriptor with Offset) writes *nbyte* bytes from *buf* to the file associated with

file_descriptor.[↩](#)

- [pwrite64\(\)](#) (Write to Descriptor with Offset (large file enabled)) writes *nbyte* bytes from *buf* to the file associated with *file_descriptor*.[↩](#)

<i>Other Functions that Operate on Files</i>	
Function	Description
givedescriptor()	Give file access to another job Give socket access to another job
select()	Check I/O status of multiple file descriptors Wait for events on multiple sockets
takedescriptor()	Take file access from another job Take socket access from another job

Note: These functions use header (include) files from the library QSYSINC, which is optionally installable. Make sure QSYSINC is installed on your system before using any of the functions. See [Header Files for UNIX-Type Functions](#)) for the file and member name of each header file.

Many of the terms used in this chapter, such as current directory, file system, path name, and link, are explained in the [Integrated File System](#) book. The API [Examples](#) also shows an example of using several integrated file system functions.

To determine whether a particular function updates the access, change, and modification times of the object on which it performs an operation, see [Integrated File System APIs--Time Stamp Updates in Volume 2](#).

access()--Determine File Accessibility

Syntax

```
#include <unistd.h>

int access(const char *path, int amode);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **access()** function determines whether a file can be accessed in a particular manner. When checking whether a job has appropriate permissions, **access()** looks at the *real* user ID (uid) and group ID (gid), not the effective IDs. Adopted authority is not used.

Parameters

path

(Input) A pointer to the null-terminated path name for the file to be checked for accessibility.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

const char **path* is the name of the file whose accessibility you want to determine. If the named file is a symbolic link, **access()** resolves the symbolic link.

See [QlgAccess-- Determine File Accessibility \(using NLS-enabled path name\)](#) for a description and an example of supplying the *path* in any CCSID.

amode

(Input) A bitwise representation of the access permissions to be checked.

The following symbols, which are defined in the <**unistd.h**> header file, can be used in *amode*:

F_OK

Tests whether the file exists

R_OK

Tests whether the file can be accessed for reading

W_OK

Tests whether the file can be accessed for writing

X_OK

Tests whether the file can be accessed for execution

You can take the bitwise inclusive OR of any or all of the last three symbols to test several access

modes at once. If you are using F_OK to test for the existence of the file, you cannot use OR with any of the other symbols. If any other bits are set in *amode*, `access()` returns the [EINVAL] error.

If the job has *ALLOBJ special authority, `access()` will indicate success for R_OK, W_OK, or X_OK even if none of the permission bits are set.

Authorities

Authorization Required for access()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be tested	*X	EACCES
Object when R_OK is specified	*R	EACCES
Object when W_OK is specified	*W	EACCES
Object when X_OK is specified	*X	EACCES
Object when R_OK W_OK is specified	*RW	EACCES
Object when R_OK X_OK is specified	*RX	EACCES
Object when W_OK X_OK is specified	*WX	EACCES
Object when R_OK W_OK X_OK is specified	*RWX	EACCES
Object when F_OK is specified	None	None

Return Value

0

`access()` was successful.

-1

`access()` was not successful (the specified access is not permitted). The *errno* global variable is set to indicate the error.

Error Conditions

If `access()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may

also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[EINTR]

Interrupted function call.

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than POSIX_SYMLINK_MAX (defined in the limits.h header file). Symbolic links are encountered during resolution of the directory or path name.

[ENAMETOOLONG]

A path name is too long.

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

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

➤[ETXTBSY]

Text file busy.

An attempt was made to execute an OS/400 PASE program that is currently open for writing, or an attempt has been made to open for writing an OS/400 PASE program that is being executed.⏪

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E

Possible APAR condition or hardware failure.

CPFA0D4 E

File system error occurred. Error number &1.

CPF3CF2 E

Error(s) occurred during running of &1 API.

CPF9872 E

Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:
 - Where multiple threads exist in the job.
 - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - **»Independent ASP QSYS.LIB«**
 - QOPT

2. Network File System Differences




Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS)

command determine the time between refresh operations of local data.)

3. QOPT File System Differences

If the object exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for the object and preceding directories in the path name follows the rules described in [Figure 1-3](#), Authorization Required for access(). If the object exists on a volume formatted in some other media format, no authorization checks are made on the object or preceding directories. The volume authorization list is checked for the requested authority regardless of the volume media format.

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<limits.h>` file (see [Header Files for UNIX-Type Functions](#))
- [accessx\(\)--Determine File Accessibility for Class of Users](#) 
- [chmod\(\)--Change File Authorizations](#)
- [faccessx\(\)--Determine File Accessibility for Class of Users](#) 
- [open\(\)--Open File](#)
- [QlgAccess--Determine File Accessibility using NLS-enabled path name](#)
- [QlgAccessx\(\)--Determine File Accessibility for Class of Users \(using NLS-enabled path name\)](#) 
- [stat\(\)--Get File Information](#)

Example

The following example determines how a file is accessed:

```
#include <stdio.h>
#include <unistd.h>

main() {
    char path[] = "/";

    if (access(path, F_OK) != 0)
        printf("'%s' does not exist!\n", path);
    else {
        if (access(path, R_OK) == 0)
            printf("You have read access to '%s'\n", path);
        if (access(path, W_OK) == 0)
            printf("You have write access to '%s'\n", path);
        if (access(path, X_OK) == 0)
            printf("You have search access to '%s'\n", path);
    }
}
```

Output:

The output from a user with read and execute access is:

```
You have read access to '/'  
You have search access to '/'
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)



accessx()--Determine File Accessibility for a Class of Users

Syntax

```
#include <unistd.h>

int accessx(const char *path, int amode, int who);
Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.
```

The **accessx()** function determines whether a file can be accessed by a specified class of users in a particular manner.

The caller must have authority to all components in the path name prefix.

Adopted authority is not used.

Parameters

path

(Input) A pointer to the null-terminated path name for the file to be checked for accessibility.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

const char **path* is the name of the file whose accessibility you want to determine. If the named file is a symbolic link, **accessx()** resolves the symbolic link.

See [QlgAccessx](#)-- Determine File Accessibility for Class of Users (using NLS-enabled path name) for a description and an example of supplying the *path* in any CCSID.

amode

(Input) A bitwise representation of the access permissions to be checked.

The following symbols, which are defined in the **<unistd.h>** header file, can be used in *amode*:

F_OK

(x'00') Tests whether the file exists

R_OK

(x'04') Tests whether the file can be accessed for reading

W_OK

(x'02') Tests whether the file can be accessed for writing

X_OK

(x'01') Tests whether the file can be accessed for execution

You can take the bitwise inclusive OR of any or all of the last three symbols to test several access modes at once. If you are using *F_OK* to test for the existence of the file, you cannot use OR with any of the other symbols. If any other bits are set in *amode*, **accessx()** returns the [EINVAL] error.

who

(Input) The class of users whose authority is to be checked.

The following symbols, which are defined in the <**unistd.h**> header file, can be used in *who*:

ACC_SELF

(x'00') Determines if specified access is permitted for the current thread. The effective user and group IDs are used.

Note: If the real and effective user ID are the same and the real and effective group ID are the same, the request is treated as *ACC_INVOKER*. See the Usage Notes for more details.

ACC_INVOKER

(x'01') Determines if specified access is permitted for the current thread. The real user and group IDs are used.

Note: The expression **access(path, amode)** is equivalent to **accessx(path, amode, ACC_INVOKER)**

ACC_OTHERS

(x'08') Determines if specified access is permitted for any user other than the object owner. Only one of *R_OK*, *W_OK*, and *X_OK* is permitted when *who* is *ACC_OTHERS*. Privileged users (users with *ALLOBJ special authority) are not considered in this check.

ACC_ALL

(x'20') Determines if specified access is permitted for all users. Only one of *R_OK*, *W_OK*, and *X_OK* is permitted when **who** is *ACC_ALL*. Privileged users (users with *ALLOBJ special authority) are not considered in this check.

Authorities

Authorization Required to Path Prefix for accessx()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be tested	*X	EACCES

The following authorities are required if the **who** parameter is *ACC_SELF* or *ACC_INVOKER*. If *ACC_SELF* is specified, the effective UID and GID of the caller are used. If *ACC_INVOKER* is used, the real UID and GID of the caller are used.

Authorization Required to Object for accessx()

Object Referred to	Authority Required	errno
Object when R_OK is specified	*R	EACCES
Object when W_OK is specified	*W	EACCES
Object when X_OK is specified	*X	EACCES
Object when R_OK W_OK is specified	*RW	EACCES
Object when R_OK X_OK is specified	*RX	EACCES
Object when W_OK X_OK is specified	*WX	EACCES
Object when R_OK W_OK X_OK is specified	*RWX	EACCES
Object when F_OK is specified	None	None

If the thread has *ALLOBJ special authority, **accessx()** with *ACC_SELF* or *ACC_INVOKER* will indicate success for R_OK, W_OK, or X_OK even if none of the permission bits are set.

Return Value

0

accessx() was successful.

-1

accessx() was not successful (or the specified access is not permitted for the class of users being checked). The *errno* global variable is set to indicate the error.

Error Conditions

If **access()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

The class of users specified by the *who* parameter does not have the permission indicated by the *amode* parameter.

The thread does not have access to the specified file, directory, component, or path prefix.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[EINTR]

Interrupted function call.

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than `POSIX_SYMLoop` (defined in the `limits.h` header file). Symbolic links are encountered during resolution of the directory or path name.

[ENAMETOOLONG]

A path name is too long.

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

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[ETXTBSY]

Text file busy.

An attempt was made to execute an OS/400 PASE program that is currently open for writing, or an attempt has been made to open for writing an OS/400 PASE program that is being executed.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E

Possible APAR condition or hardware failure.

CPFA0D4 E

File system error occurred. Error number &1.

CPF3CF2 E

Error(s) occurred during running of &1 API.

CPF9872 E

Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:
 - Where multiple threads exist in the job.
 - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - Independent ASP QSYS.LIB
 - QOPT

2. ACC_SELF Mapped to ACC_INVOKER

Some physical file systems do not support *ACC_SELF* for the *who* parameter. Therefore, **accessx()** will change the *who* parameter from *ACC_SELF* to *ACC_INVOKER* if the caller's real and effective user ID are equal, and the caller's real and effective group ID are equal.

3. Network File System Differences

The Network File System will only support the value *ACC_INVOKER* for the *who* parameter. If **accessx()** is called on a file in a mounted Network File System directory with a value for *who* other than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP. **Note:** If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)

4. QNTC File System Differences

The QNTC File System will only support the value *ACC_INVOKER* for the *who* parameter. If **accessx()** is called on a file in the QNTC File System with a value for *who* other than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP. **Note:** If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

5. QOPT File System Differences

If the object exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for the object and preceding directories in the path name follows the rules described in [the previous table](#), Authorization Required to Object for **accessx()**. If the object exists on a volume formatted in some other media format, no authorization checks are made on the object or preceding directories. The volume authorization list is checked for the requested authority regardless of the volume media format.

6. QFileSvr.400 File System Differences

The QFileSvr.400 File System will only support the value *ACC_INVOKER* for the *who* parameter. If **accessx()** is called on a file in the QFileSvr.400 File System with a value for *who* other than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP. **Note:** If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

7. QNetWare File System Differences

The QNetWare File System will only support the value `ACC_INVOKER` for the `who` parameter. If `accessx()` is called on a file in the QNetWare File System with a value for `who` other than `ACC_INVOKER`, the call will return -1 and `errno` `ENOTSUP`. **Note:** If the value for `who` has been mapped from `ACC_SELF` to `ACC_INVOKER` as previously described, then `ENOTSUP` will not be returned.

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<limits.h>` file (see [Header Files for UNIX-Type Functions](#))
- [chmod\(\)](#)--Change File Authorizations
- [open\(\)](#)--Open File
- [access\(\)](#)--Determine File Accessibility
- [faccessx\(\)](#)--Determine File Accessibility for a Class of Users
- [QlgAccessx\(\)](#)--Determine File Accessibility for a Class of Users (using NLS-enabled path name)
- [QlgAccess\(\)](#)--Determine File Accessibility (using NLS-enabled path name)
- [stat\(\)](#)--Get File Information

Example

The following example determines how a file is accessed:

```
#include <stdio.h>
#include <unistd.h>

main() {
    char path[]="/myfile";

    if (accessx(path, R_OK, ACC_OTHERS) == 0)
        printf("Someone besides the owner has read access to '%s'\n", path);
    if (accessx(path, W_OK, ACC_OTHERS) == 0)
        printf("Someone besides the owner has write access to '%s'\n", path);
    if (accessx(path, X_OK, ACC_OTHERS) == 0)
        printf("Someone besides the owner has search access to '%s'\n", path);
}
```

Output:

In this example `accessx()` was called on `/'myfile'`. The following would be the output if someone other than the

owner has *R authority, someone besides the owner has *W authority, and noone other than the owner has *X authority.

Someone besides the owner has read access to '/'
Someone besides the owner has write access to '/'



API introduced: V5R2

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

chdir()--Change Current Directory

Syntax

```
#include <unistd.h>

int chdir(const char *path);
Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.
```

The **chdir()** function makes the directory named by *path* the new current directory. If the last component of *path* is a symbolic link, **chdir()** resolves the contents of the symbolic link. If the **chdir()** function fails, the current directory is unchanged.

Parameters

path

(Input) A pointer to the null-terminated path name of the directory that should become the current directory.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See [QlgChdir\(\)--Change Current Directory](#) for a description and an example of supplying the *path* in any CCSID.

Authorities

Note: Adopted authority is not used.

Authorization Required for chdir()

Object Referred to	Authority Required	errno
Each directory of the path name	*X	EACCES

Return Value

0

chdir() was successful.

-1

chdir() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **chdir()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than POSIX_SYMLINK_MAX (defined in the limits.h header file). Symbolic links are encountered during resolution of the directory or path name.

[ENAMETOOLONG]

A path name is too long.

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

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E

Possible APAR condition or hardware failure.

CPFA0D4 E

File system error occurred. Error number &1.

CPF3CF2 E

Error(s) occurred during running of &1 API.

CPF9872 E

Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - [»Independent ASP QSYS.LIB«](#)
 - QOPT

The **chdir()** API operates on two objects: the previous current working directory and the new one. If either of these objects is managed by a file system that is not threadsafe, **chdir()** fails with the ENOTSAFE error code.

2. QOPT File System Differences

If the directory exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for each directory in the path name follows the rules described in [Authorization Required for chdir\(\)](#). If the directory exists on a volume formatted in some other media format, no authorization checks are made on each directory in the path name. The volume authorization list is checked for *USE authority regardless of the volume media format.

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<limits.h>` file (see [Header Files for UNIX-Type Functions](#))
- [»fchdir\(\)--Change Current Directory by Descriptor«](#)
- [getcwd\(\)--Get Current Directory](#)

- [QlgChdir\(\)--Change Current Directory](#)
- [»QlgGetcwd\(\)--Get Current Directory«](#)

Example

The following example uses **chdir()**:

```
#include <stdio.h>
#include <unistd.h>

main() {
    if (chdir("/tmp") != 0)
        perror("chdir() to /tmp failed");
    if (chdir("/chdir/error") != 0)
        perror("chdir() to /chdir/error failed");
}
```

Output:

```
chdir() to /chdir/error failed: No such path or directory.
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

chmod()--Change File Authorizations

Syntax

```
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see [Usage Notes](#).

The **chmod()** function changes [S_ISUID](#), [S_ISGID](#), and the permission bits of the file or directory specified in *path* to the corresponding bits specified in *mode*. [If](#) the named file is a symbolic link, **chmod()** resolves the symbolic link. **chmod()** has no effect on file descriptions for files that are open at the time **chmod()** is called.

When **chmod()** is successful it updates the change time of the file.

If the file is checked out by another user (someone other than the user profile of the current job), **chmod()** fails with the [EBUSY] error.

Parameters

path

(Input) A pointer to the null-terminated path name of the file whose mode is being changed.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See [QlgChmod\(\)--Change File Authorizations](#) for a description and an example of supplying the *path* in any CCSID.

mode

(Input) Bits that define [S_ISUID](#), [S_ISGID](#), and [the](#) access permissions of the file.

The *mode* argument is created with one of the following symbols defined in the `<sys/stat.h>` include file.

S_IRUSR

Read permission for the file owner

S_IWUSR

Write permission for the file owner

S_IXUSR

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

S_IRWXU

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

S_IRGRP

Read permission for the file's group

S_IWGRP

Write permission for the file's group

S_IXGRP

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

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_IROTH

General read permission

S_IWOTH

General write permission

S_IXOTH

General search permission (for a directory) or general execute permission (for a file)

S_IRWXO

General read, write, and search or execute permission. *S_IRWXO* is the bitwise inclusive OR of *S_IROTH*, *S_IWOTH*, and *S_IXOTH*

S_ISUID

Set effective user ID at execution time. This bit is ignored if the object specified by *path* is a directory.

S_ISGID

Set effective group ID at execution time. [»See Usage Notes](#) for more information [«](#)if the object specified by *path* is a directory.

If bits other than the bits listed above are set in *mode*, **chmod()** returns the [EINVAL] error.

Authorities

Note: Adopted authority is not used.

Authorization required for chmod() (excluding QDLS)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object	Owner (see Note)	EPERM
Note: You do not need the listed authority if you have *ALLOBJ special authority.		

Authorization required for chmod() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object	Owner or *ALL	EACCES

Return Value

0

chmod() was successful.

-1

chmod() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **chmod()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

» *[EJRNDAMAGE]*

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were

attempting to send an entry to the journal.

[EJRNENTTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.

[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal. <<

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than POSIX_SYMLOOP (defined in the limits.h header file). Symbolic links are encountered during resolution of the directory or path name.

[ENAMETOOLONG]

A path name is too long.

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

>>*[ENEWJRN]*

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal. <<

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[EPERM]

Operation not permitted.

You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.

The thread does not have authority to perform the requested function.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E

Possible APAR condition or hardware failure.

CPFA0D4 E

File system error occurred. Error number &1.

CPF3CF2 E

Error(s) occurred during running of &1 API.

CPF9872 E

Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - [»Independent ASP QSYS.LIB «](#)
 - QOPT

2. Root, QOpenSys, and User-Defined File System Differences

If the object has a primary group, it must match the primary group ID or one of the supplemental group IDs of the caller of the API; otherwise, the S_ISGID bit is turned off.

3. QSYS.LIB [»](#) and independent ASP QSYS.LIB [«](#) File System Differences

chmod() is not supported for member (.MBR) objects.

chmod() returns [EBUSY] if the object is allocated in another job.

QSYS.LIB [»](#) and independent ASP QSYS.LIB do [«](#)not support setting the S_ISUID (set-user-ID) and S_ISGID (set-group_ID) bits. If they are turned on in the mode parameter, they are ignored.

4. QDLS File System Differences

Changing the permissions of the /QDLS directory (the root folder) is not allowed. If an attempt is made to change the permissions, an [ENOTSUP] error is returned.

"Group" rights are not set if there is no current group.

QDLS does not support setting the S_ISUID and S_ISGID bits. If they are turned on in the mode parameter, they are ignored.

5. QOPT File System Differences

Changing the permissions is allowed only for an object that exists on a volume formatted in Universal Disk Format (UDF). For all other media formats, ENOTSUP is returned.

In addition to the authorization checks described in [Authorization Required for chmod\(\)](#), the volume authorization list is checked for *CHANGE authority.

QOPT does not support setting the S_ISUID and S_ISGID bits for any optical media format. If they are turned on in the mode parameter, ENOTSUP is returned.

6. QNetWare File System Differences

The QNetWare file system does not fully support **chmod()**. See [NetWare on iSeries](#) for more information.

QNetWare supports the S_ISUID and S_ISGID bits by passing them to the server and surfacing them to the caller. Some versions of NetWare may support the bits and others may not.

7. QFileSvr.400 Differences

QFileSvr.400 supports the S_ISUID and S_ISGID bits by passing them to the server and surfacing them to the caller.


8. Network File System Differences

The NFS client supports the S_ISUID and S_ISGID bits by passing them to the server over the network and surfacing them to the caller. Whether a particular network file system supports the setting of these bits depends on the server. Most servers have the capability of masking off these bits if the NOSUID option is specified on the export. The default, however, is to support these bits.

9. QNTC File System Differences

chmod() does not update the Windows NT server access control lists that control the authority of users to the file or directory. The mode settings are ignored.

10. S_ISGID bit of a directory in Root, QOpenSys, or User-Defined File System

The S_ISGID bit of the directory affects what the group ID (GID) is for objects that are created in the directory. If the S_ISGID bit of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the object. If the S_ISGID bit of the parent directory is on, the group ID (GID) of the new object is set to the GID of the parent directory. For all other file systems, the GID of the new object is set to the GID of the parent directory. 

Related Information

- The `<sys/types.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<sys/stat.h>` file (see [Header Files for UNIX-Type Functions](#))
- [chown\(\)--Change Owner and Group of File](#)
- [fchmod\(\)--Change File Authorizations by Descriptor](#)
- [mkdir\(\)--Make Directory](#)
- [open\(\)--Open File](#)
- [stat\(\)--Get File Information](#)
- See [QlgChmod\(\)--Change File Authorizations](#)

Example

The following example changes the permissions for a file:

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>

main() {
    char fn[]="temp.file";
    int file_descriptor;
    struct stat info;

    if ((file_descriptor = creat(fn, S_IWUSR)) == -1)
        perror("creat() error");
    else {
        if (stat(fn, &info) != 0)
            perror("stat() error");
        else {
            printf("original permissions were: %08o\n", info.st_mode);
        }
        if (chmod(fn, S_IRWXU|S_IRWXG) != 0)
            perror("chmod() error");
        else {
            if (stat(fn, &info) != 0)
                perror("stat() error");
            else {
                printf("after chmod(), permissions are: %08o\n", info.st_mode);
            }
        }
    }
}
```

```
    if (close(file_descriptor) != 0)
        perror("close() error");
    if (unlink(fn) != 0)
        perror("unlink() error");
}
}
```

Output:

```
original permissions were: 00100200
after chmod(), permissions are: 00100770
```

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

chown()--Change Owner and Group of File

Syntax

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

```
int chown(const char *path, uid_t owner, gid_t group);
```

Threadsafe: Conditional; see Usage Notes.

The **chown()** function changes the owner and group of a file. If the named file is a symbolic link, **chown()** resolves the symbolic link. The permissions of the previous owner or primary group to the object are revoked.

If the file is checked out by another user (someone other than the user profile of the current job), **chown()** fails with the [EBUSY] error.

When **chown()** completes successfully, it updates the change time of the file.

Parameters

path

(Input) A pointer to the null-terminated path name of the file whose owner and group are being changed.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See [QlgChown\(\)--Change Owner and Group of File](#) for a description and an example of supplying the *path* in any CCSID.

owner

(Input) The user ID (uid) of the new owner of the file.

group

(Input) The group ID (gid) of the new group for the file.

Note: Changing the owner or the primary group causes the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode to be cleared, unless the caller has *ALLOBJ special authority. If the caller does have *ALLOBJ special authority, the bits are not changed. This does not apply to directories or FIFO special files. See the [chmod\(\)](#) documentation.

Authorities

Note: Adopted authority is not used.

Authorization Required for chown() (excluding QSYS.LIB, independent ASP QSYS.LIB, and QDLS)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object, when changing the owner	Owner and *OBJEXIST (also see Note 1)	EPERM
Object, when changing the primary group	See Note 2	EPERM
Previous owner's user profile, when changing the owner	*DLT	EPERM
New owner's user profile, when changing the owner	*ADD	EPERM
User profile of previous primary group, when changing the primary group	*DLT	EPERM
New primary group's user profile, when changing the primary group	*ADD	EPERM


Note:

1. You do not need the listed authority if you have *ALLOBJ special authority.
2. At least one of the following must be true:
 - a. You have *ALLOBJ special authority.
 - b. You are the owner and either of the following:
 - The new primary group is the primary group of the job.
 - The new primary group is one of the supplementary groups of the job.

Authorization Required for chown() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X See Note 1	EACCES
Object when changing the owner	See Note 2(a)	EPERM
Object when changing the primary group	See Note 2(b)	EPERM

Note:

1. For *FILE objects (such as DDM file, diskette file, print file, and save file), *RX authority is required to the parent directory of the object, rather than just *X authority.
2. The required authorization varies for each object type. For details of the following commands, see the [iSeries Security Reference](#)  book.
 - a. CHGOWN
 - b. CHGPGP

Authorization Required for chown() in the QDLS File System

Object Referred to	Authority Required	errno

Each directory in the path name preceding the object	*X	EACCES
Object	*ALLOBJ Special Authority or Owner	EPERM
Previous owner's user profile, when changing the owner	*DLT	EPERM
New owner's user profile, when changing the owner	*ADD	EPERM
Previous primary group's user profile, when changing the primary group	*DLT	EPERM
New primary group's user profile, when changing the primary group	*ADD	EPERM

Authorization Required for `chown()` in the QOPT File System

Object Referred to	Authority Required	errno
Volume authorization list	*CHANGE	EACCES
Each directory in the path name preceding the object.	*X	EACCES
Object	*ALLOBJ Special Authority or Owner	EPERM

Return Value

0

`chown()` was successful.

-1

`chown()` was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If `chown()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

owner or *group* is not a valid user ID (uid) or group ID (gid).

owner is the current primary group of the object.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

➤*[EJRNDAMAGE]*

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.

[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal. ❄

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than POSIX_SYMLOOP (defined in the limits.h header file). Symbolic links are encountered during resolution of the directory or path name.

[ENAMETOOLONG]

A path name is too long.

A path name is longer than PATH_MAX characters or some component of the name is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length

of the name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using the **pathconf()** function.

»[ENEWJRN]

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal. «

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[EPERM]

Operation not permitted.

You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.

The thread does not have authority to perform the requested function.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

Error Messages

The following messages may be sent from this API:

CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

- Root
- QOpenSys
- User-defined
- QNTC
- QSYS.LIB
- »Independent ASP QSYS.LIB «
- QOPT

2. QSYS.LIB »and Independent ASP QSYS.LIB «File System Differences

chown() is not supported for member (.MBR) objects.

3. QDLS File System Differences

The owner and primary group of the /QDLS directory (root folder) cannot be changed. If an attempt is made to change the owner and primary group, a [ENOTSUP] error is returned.

4. QOPT File System Differences

Changing the owner and primary group is allowed only for an object that exists on a volume formatted in Universal Disk Format (UDF). For all other media formats, ENOTSUP will be returned.

QOPT file system objects that have owners will not be recognized by the Work with Objects by Owner (WRKOBJOWN) CL command. Likewise, QOPT objects that have a primary group will not be recognized by the Work Objects by Primary Group) CL command.

5. QFileSvr.400 File System Differences

The QFileSvr.400 file system does not support **chown()**.

6. QNetWare File System Differences

The QNetWare file system does not support primary group. The GID must be zero.

7. QNTC File System Differences

The owner of files and directories cannot be changed. All files and directories in QNTC are owned by the QDFTOWN user profile.

Related Information

- The <[unistd.h](#)> file (see [Header Files for UNIX-Type Functions](#))
- The <[limits.h](#)> file (see [Header Files for UNIX-Type Functions](#))
- [chmod\(\)--Change File Authorizations](#)
- [fchown\(\)--Change Owner and Group of File by Descriptor](#)
- [fstat\(\)--Get File Information by Descriptor](#)
- [lstat\(\)--Get File or Link Information](#)
- [stat\(\)--Get File Information](#)
- [OlgChown\(\)--Change Owner and Group of File](#)

Example

The following example changes the owner and group of a file:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

main() {
    char fn[]="temp.file";
    int file_descriptor;
    struct stat info;

    if ((file_descriptor = creat(fn, S_IRWXU)) == -1)
        perror("creat() error");
    else {
        close(file_descriptor);
        stat(fn, &info);
        printf("original owner was %d and group was %d\n", info.st_uid,
            info.st_gid);
        if (chown(fn, 152, 0) != 0)
            perror("chown() error");
        else {
            stat(fn, &info);
            printf("after chown(), owner is %d and group is %d\n",
                info.st_uid, info.st_gid);
        }
        unlink(fn);
    }
}
```

Output:

```
original owner was 137 and group was 0  
after chown(), owner is 152 and group is 0
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

close()--Close File or Socket Descriptor

Syntax

```
#include <unistd.h>

int close(int fd);
```

Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see [Usage Notes](#).

The **close()** function closes a descriptor, *fd*. This frees the descriptor to be returned by future **open()** calls and other calls that create descriptors.

When the last open descriptor for a file is closed, the file itself is closed. If the link count of the file is zero at that time, the space occupied by the file is freed and the file becomes inaccessible.

close() unlocks (removes) all outstanding byte locks that a job has on the associated file.

When all file descriptors associated with a pipe or FIFO special file are closed, any data remaining in the pipe or FIFO is discarded and internal storage used is returned to the system.

When *fd* refers to a socket, **close()** closes the socket identified by the descriptor.

Parameters

fd

(Input) The descriptor to be closed.

Authorities

No authorization is required. Authorization is verified during **open()**, **creat()**, or **socket()**.

Return Value

close() returns an integer. Possible values are:

0 **close()** was successful.

-1 **close()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If `close()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADF]

Descriptor not valid.

A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

➤*[EJRNDAMAGE]*

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.

[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal.

[ENEWJRN]

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal.⏪

[ENOBUFFS]

There is not enough buffer space for the requested operation.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOSYS]

Function not implemented.

An attempt was made to use a function that is not available in this implementation for any object or any arguments.

The path name given refers to an object that does not support this function.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

- Root
- QOpenSys
- User-defined
- QNTC
- QSYS.LIB
- »Independent ASP QSYS.LIB«
- QOPT

2. When a socket descriptor is closed, the system tries to send any queued data associated with the socket.

- For AF_NS or AF_INET sockets, depending on whether the SO_LINGER socket option is set, queued data may be discarded.

Note: For these sockets, the default value for the SO_LINGER socket option has the option flag set off (the system attempts to send any queued data with an infinite wait time).

- For AF_TELEPHONY sockets, depending on whether the SO_LINGER socket option is set, buffered data may be discarded.

Note: For these sockets, the default value for the SO_LINGER socket option has the option flag set on with a time value of 1 second (the system will wait up to 1 second to send buffered data before clearing the telephone connection).

3. A socket descriptor being shared among multiple processes is not closed until the process that issued the *close()* is the last process with access to the socket.

Related Information

- The <unistd.h> file (see [Header Files for UNIX-Type Functions](#))
- [creat\(\)](#)--Create or Rewrite File
- [dup\(\)](#)--Duplicate Open File Descriptor
- [dup2\(\)](#)--Duplicate Open File Descriptor to Another Descriptor
- [fcntl\(\)](#)--Perform File Control Command
- [open\(\)](#)--Open File
- [setsockopt\(\)](#)--Set Socket Options
- [unlink\(\)](#)--Remove Link to File

Example

The following example uses `close()`

See [Code disclaimer information](#) for information pertaining to code examples.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

main() {
    int fd1, fd2;
    char out[20]="Test string",
        fn[]="test.file",
        in[20];
    short write_error;

    memset(in, 0x00, sizeof(in));

    write_error = 0;

    if ( (fd1 = creat(fn,S_IRWXU)) == -1)
        perror("creat() error");
    else if ( (fd2 = open(fn,O_RDWR)) == -1)
        perror("open() error");
    else {
        if (write(fd1, out, strlen(out)+1) == -1) {
            perror("write() error");
            write_error = 1;
        }
        close(fd1);
        if (!write_error) {
            if (read(fd2, in, sizeof(in)) == -1)
                perror("read() error");
            else printf("string read from file was: '%s'\n", in);
        }
        close(fd2);
    }
}
```

Output:

```
string read from file was: 'Test string'
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

closedir()--Close Directory

Syntax

```
#include <sys/types.h>
#include <dirent.h>

int closedir(DIR *dirp);
Threadsafe: Conditional; see Usage Notes.
```

The **closedir()** function closes the directory stream indicated by *dirp*. It frees the buffer that **readdir()** uses when reading the directory stream.

A file descriptor is used for type DIR; **closedir()** closes the file descriptor.

Parameters

dirp

(Input) A pointer to a DIR that refers to the open directory stream to be closed. This pointer is returned by the **opendir()** function.

Authorities

No authorization is required. Authorization is verified during **opendir()**.

Return Value

0

closedir() was successful.

-1

closedir() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **closedir()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADF]

Descriptor not valid.

A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file.

This may occur when *dirp* does not refer to an open directory stream.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

This may occur when *dirp* does not refer to an open directory stream.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

➤ *[EJRNDDAMAGE]*

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNENTTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.

[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal.

[ENEWJRN]

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal.❄

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOSYS]

Function not implemented.

An attempt was made to use a function that is not available in this implementation for any object or any arguments.

The path name given refers to an object that does not support this function.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E

Possible APAR condition or hardware failure.

CPFA0D4 E

File system error occurred. Error number &1.

CPF3CF2 E

Error(s) occurred during running of &1 API.

CPF9872 E

Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - [»Independent ASP QSYS.LIB «](#)
 - QOPT
2. If the *dirp* argument passed to **closedir()** does not refer to an open directory, **closedir()** returns the [EBADF] or [EFAULT] error.
3. After a call to **closedir()** the *dirp* will not point to a valid DIR.

Related Information

- The `<sys/types.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<dirent.h>` file (see [Header Files for UNIX-Type Functions](#))
- [opendir\(\)--Open Directory](#)
- [readdir\(\)--Read Directory Entry](#)
- [rewinddir\(\)--Reset Directory Stream to Beginning](#)

Example

The following example closes a directory:

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

main() {
    DIR *dir;
    struct dirent *entry;
    int count;

    if ((dir = opendir("/")) == NULL)
        perror("opendir() error");
    else {
        count = 0;
        while ((entry = readdir(dir)) != NULL) {
```

```
        printf("directory entry %03d: %s\n", ++count, entry->d_name);
    }
    closedir(dir);
}
}
```

Output:

```
directory entry 001: .
directory entry 002: ..
directory entry 003: QSYS.LIB
directory entry 004: QDLS
directory entry 005: QOpenSys
directory entry 006: home
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

creat()--Create or Rewrite File

Syntax

```
#include <fcntl.h>

int creat(const char *path, mode_t mode);
Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.
```

The **creat()** function creates a new file or rewrites an existing file so that it is truncated to zero length. The function call

```
creat(path, mode);
```

is equivalent to the call

```
open(path, O_CREAT|O_WRONLY|O_TRUNC, mode);
```

This means that the file named by *path* is created if it does not already exist, opened for writing only, and truncated to zero length. For further information, see [open\(\)--Open File](#).

The *mode* argument specifies file permission bits to be used in creating the file. For more information on *mode*, see [chmod\(\)--Change File Authorizations](#).

Parameters

path

(Input) A pointer to the null-terminated path name of the file to be created or rewritten.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

When a new file is created, the new file name is assumed to be represented in the language and country or region currently in effect for the job.

See [QlgCreat\(\)--Create or Rewrite File](#) for a description and an example of supplying the *path* in any CCSID.

mode

(Input) The file permission bits to be used when creating the file. The S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits also may be specified when creating the file.

See [chmod\(\)--Change File Authorizations](#) for details on the values that can be specified for

mode.

Authorities

Note: Adopted authority is not used.

Figure 1-11. Authorization Required for creat() (excluding QSYS.LIB, independent ASP QSYS.LIB, and QDLS)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be created	*X	EACCES
Existing object	*W	EACCES
Parent directory of object to be created when object does not exist	*WX	EACCES

Figure 1-12. Authorization Required for creat() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be created	*X	EACCES
Existing object	*W	EACCES
Parent directory of object to be created when object does not exist	*OBJMGT or *OBJALTER	EACCES
Parent directory of the parent directory of object to be created when object does not exist	*Add	EACCES

Figure 1-13. Authorization Required for creat() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be created	*X	EACCES
Existing object	*W	EACCES
Parent directory of object to be created when object does not exist	*CHANGE	EACCES

Return Value

value

creat() was successful. The value returned is the file descriptor for the open file.

-1

creat() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If `creat()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

The open sharing mode may conflict with another open of this file, or `O_WRONLY` or `O_RDWR` is specified and the file is checked out by another user.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EEXIST]

File exists.

The file specified already exists and the specified operation requires that it not exist.

The named file, directory, or path already exists.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

- Unused bits in *mode* are set and should be cleared.
- It is invalid to open this type of object.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[EISDIR]

Specified target is a directory.

The path specified named a directory where a file or object name was expected.

The path name given is a directory.

➤[EJRNDDAMAGE]

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNENTTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.


[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal. 

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than POSIX_SYMLOOP (defined in the limits.h header file). Symbolic links are encountered during resolution of the directory or path name.

[EMFILE]

Too many open files for this process.

An attempt was made to open more files than allowed by the value of OPEN_MAX. The value of OPEN_MAX can be retrieved using the sysconf() function.

The process has more than OPEN_MAX descriptors already open (see the **sysconf()** function).

[ENAMETOOLONG]

A path name is too long.

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

 *[ENEWJRN]*

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal.❏

[ENFILE]

Too many open files in the system.

A system limit has been reached for the number of files that are allowed to be concurrently open in the system.

The entire system has too many other file descriptors already open.

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOSYSRSC]

System resources not available to complete request.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[EOVERFLOW]

Object is too large to process.

The object's data size exceeds the limit allowed by this function.

The specified file exists and its size is too large to be represented in a variable of type `off_t` (the file is larger than 2GB minus 1 byte).

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code *[ENOTSAFE]* when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

- Root
- QOpenSys
- User-defined
- QNTC
- QSYS.LIB
- »Independent ASP QSYS.LIB «
- QOPT

2. Root, QOpenSys, and User-Defined File System Differences

The user who creates the file becomes its owner. »If the S_ISGID bit of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the object. If the S_ISGID bit of the parent directory is on, the «group ID (GID) is copied from the parent directory in which the file is created.

The owner, primary group, and public object authorities (*OBJEXIST, *OBJMGT, *OBJALTER, and *OBJREF) are copied from the parent directory's owner, primary group, and public object authorities. This occurs even when the new file has a different owner than the parent directory. The owner, primary group, and public data authorities (*R, *W, and *X) are derived from the permissions specified in the mode (except for those permissions that are also set in the file mode creation mask). The new file does not have any private authorities or authorization list. It only has authorities for the owner, primary group, and public.

3. QSYS.LIB »and Independent ASP QSYS.LIB «File System Differences

When a new member is created, the mode and profiles must match those of the parent file. If they do not match, the create operation will fail.

The user who creates a member becomes the owner of the member. However, this owner must be the same as the owner of the parent directory in which the member is being created.

The group ID is obtained from the primary user profile, if one exists. This group ID must be the same as the group ID of the file in which the member is being created.

The owner object authorities are set to *OBJEXIST, *OBJMGT, *OBJALTER, and *OBJREF. The primary group object authorities are specified by options in the user profile of the job. None of the public object authorities are set.

The owner, primary group, and public data authorities (*R, *W, and *X) are derived from the permissions specified in the mode (except for those permissions that are also set in the file mode creation mask). The data authorities must match the data authorities of the file in which the member is being created.

The primary group authorities are not saved if the primary group does not exist. When a primary group is attached to the object, the object gets the authorities specified in *mode*.

A member cannot be created in a mixed-CCSID file.

The file access time for a database member is updated using the normal rules that apply to database files. At most, the access time is updated once per day.

4. QDLS File System Differences

The user who creates the file becomes its owner. The group ID is copied from the parent folder in which the file is created.

The owner object authority is set to *OBJMGT + *OBJEXIST + *OBJALTER + *OBJREF.

The primary group and public object authority and all other authorities are copied from the parent folder.

The owner, primary group, and public data authority (including *OBJOPR) are derived from the permissions specified in *mode* (except those permissions that are also set in the file mode creation mask).

The primary group authorities specified in *mode* are not saved if no primary group exists.

5. QOPT File System Differences

When the volume on which the file is being created is formatted in Universal Disk Format (UDF):

- The authorization that is checked for the object and preceding directories in the path name follows the rules described in [Figure 1-11](#) "Authorization Required for creat()."
- The volume authorization list is checked for *CHANGE authority.
- The user who creates the file becomes its owner.
- The group ID is copied from the parent directory in which the file is created.
- The owner, primary group, and public data authorities (*R, *W, and *X) are derived from the permissions specified in the mode (except those permissions that are also set in the file mode creation mask).
- The resulting share mode is O_SHARE_NONE; therefore, the function call

```
creat(path, mode);
```

is equivalent to the call

```
open(path,  
      O_CREAT | O_WRONLY | O_TRUNC | O_SHARE_NONE,  
      mode);
```

- The same uppercase and lowercase forms in which the names are entered are preserved. No distinction is made between uppercase and lower case when searching for names.

When the volume on which the file is being created is not formatted in Universal Disk Format (UDF):

- No authorization checks are made on the object or preceding directories in the path name.
- The volume authorization list is checked for *CHANGE authority.
- QDFTOWN becomes the owner of the file.
- No group ID is assigned to the file.
- The permissions specified in the mode are ignored. The owner, primary group, and public data authorities are set to RWX.
- For newly created files, names are created in uppercase. No distinction is made between uppercase and lowercase when searching for names.

A file cannot be created as a direct child of /QOPT.

The change and modification times of the parent directory are not updated.

6. Network File System Differences

Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. The creation of a file may fail if permissions and other attributes that are stored locally by the Network File System are more restrictive than those at the server. A later attempt to create a file can succeed when the locally stored data has been refreshed. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) The creation can also succeed after the file system has been remounted.

If you try to re-create a file that was recently deleted, the request may fail because data that was stored locally by the Network File System still has a record of the file's existence. The creation succeeds when the locally stored data has been updated.

Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations.

7. QNetWare File System Differences

The user who creates the file becomes the owner. Mode bits are not fully supported. See [NetWare on iSeries](#) for more information.

8. This function will fail with the [Eoverflow] error if the specified file exists and its size is too large to be represented in a variable of type off_t (the file is larger than 2GB minus 1 byte).
9. When you develop in C-based languages and this function is compiled with the `_LARGE_FILES` macro defined, it will be mapped to `creat64()`.

Related Information

- The `<fcntl.h>` file (see [Header Files for UNIX-Type Functions](#))
- [creat64\(\)--Create or Rewrite a File \(Large File Enabled\)](#)
- [open\(\)--Open File](#)
- [QlgCreat\(\)--Create or Rewrite File](#)

Example

The following example creates a file:

```
#include <stdio.h>
#include <fcntl.h>

main() {
    char fn[]="creat.file", text[]="This is a test";
    int fd, rc;

    if ((fd = creat(fn, S_IRUSR | S_IWUSR)) < 0)
        perror("creat() error");
    else {
        if (-1==(rc=write(fd, text, strlen(text))))
            perror("write() error");
        if (close(fd) != 0)
            perror("close() error");
        if (unlink(fn)!= 0)
            perror("unlink() error");
    }
}
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

creat64()--Create or Rewrite a File (Large File Enabled)

Syntax

```
#include <fcntl.h>

int creat64(const char *path, mode_t mode);
```

Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **creat64()** function creates a new file or rewrites an existing file so that it is truncated to zero length. The open file instance created with **creat64()** is allowed to be larger than 2GB minus 1 byte. The function call

```
creat64(path, mode);
```

is equivalent to the call

```
open64(path, O_CREAT|O_WRONLY|O_TRUNC, mode);
```

If the file named by *path* does not already exist, it is created. The file is then opened for writing only and truncated to zero length. For further information, see [open64\(\)--Open File \(Large File Enabled\)](#).

See [QlgCreat64--Create or Rewrite a File \(Large File Enabled\)](#) for a description and an example of supplying the *path* in any CCSID.

The *mode* argument specifies file permission bits to be used in creating the file. For more information on *mode*, see [chmod\(\)--Change File Authorizations](#).

For additional information about parameters, authorities required, error conditions, and examples, see [creat\(\)--Create or Rewrite File](#).

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **creat64()** API, you must compile the source with `_LARGE_FILE_API` macro defined.
2. All of the usage notes for **creat()** apply to **creat64()**. See [Usage Notes](#) in the **creat()** API.

API introduced: V4R4

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

DosSetFileLocks()--Lock and Unlock a Byte Range of an Open File

Syntax

```
#define INCL_DOSERRORS
#define INCL_DOSFILEMGR
#include <os2.h>

APIRET APIENTRY DosSetFileLocks(HFILE FileHandle,
                                PFILELOCK ppUnlockRange,
                                PFILELOCK ppLockRange,
                                ULONG ulTimeOut,
                                ULONG ulFlags);
```

Threadsafe: Conditional; see Usage Notes.

The **DosSetFileLocks()** function locks and unlocks a range of an open file. A non-zero range indicates that a lock or unlock request is being made.

Parameters

FileHandle

(Input) The file descriptor of the file in which a range is to be locked.

ppUnlockRange.

(Input) Address of the structure containing the offset and length of a range to be unlocked. The structure is as follows:

FileOffset

The offset to the beginning of the range to be unlocked.

RangeLength

The length of the range to be unlocked. A value of zero means that unlocking is not required.

ppLockRange

(Input) Address of the structure containing the offset and length of a range to be locked. The structure is as follows:

FileOffset

The offset to the beginning of the range to be locked.

RangeLength

The length of the range to be locked. A value of zero means that locking is not required.

ulTimeOut

(Input) The maximum time, in milliseconds, that the process is to wait for the requested locks.

ulFlags

(Input) Flags that describe the action to be taken. If any flags other than those listed below are

specified, the error `ERROR_INVALID_PARAMETER` will be returned.

The following values are to be specified in *ulFlags*:

'0x0002' or `QPOL_DOSSETFILELOCKS_ATOMIC`

Atomic. This bit defines a request for atomic locking. If this bit is set to 1 and the lock range is equal to the unlock range, an atomic lock occurs. If this bit is set to 1 and the lock range is not equal to the unlock range, `ERROR_LOCK_VIOLATION` is returned.

'0x0001' or `QPOL_DOSSETFILELOCKS_SHARE`

Share. This bit defines the type of access that other processes may have to the file range that is being locked.

If this bit is set to 0 (the default), other processes have no access to the locked file range. The current process has exclusive access to the locked file range, which must not overlap any other locked file range.

If this bit is set to 1, the current process and other processes have shared access to the locked file range. A file range with shared access may overlap any other file range with shared access, but must not overlap any other file range with exclusive access.

Authorities

No authorization is required.

Return Value

`NO_ERROR (0)`

`DosSetFileLocks()` was successful.

value

When *value* is not `NO_ERROR` (non-zero), `DosSetFileLocks()` was not successful. The *value* that is returned indicates the error.

Error Conditions

If `DosSetFileLocks()` is not successful, the value that is returned is one of the following errors. The `<bseerr.h>` header file defines these values.

`[ERROR_GENERAL_FAILURE]`

A general failure occurred.

This may result from damage in the system. Refer to messages in the job log for other possible causes.

`[ERROR_INVALID_HANDLE]`

An invalid file handle was found.

The file handle passed to this function is not valid.

`[ERROR_LOCK_VIOLATION]`

A lock violation was found.

The requested lock and unlock ranges are both zero.

[ERROR_INVALID_PARAMETER]

An invalid parameter was found.

A parameter passed to this function is not valid.

The byte range specified by the offset and length in the ppUnlockRange or ppLockRange parameters extends beyond 2GB minus 1 byte.

[ERROR_ATOMIC_LOCK_NOT_SUPPORTED]

The atomic lock operation is not supported.

The file system does not support atomic lock operations.

[ERROR_TIMER_NOT_SUPPORTED]

The lock timer value is not supported.

The file system does not support the lock timer value.

Error Messages

The system may send the following messages from this function.

CPE3418 E

Possible APAR condition or hardware failure.

CPFA0D4 E

File system error occurred. Error number &1.

CPF3CF2 E

Error(s) occurred during running of &1 API.

CPF9872 E

Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code *[ERROR_GEN_FAILURE]* when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB

- [»Independent ASP QSYS.LIB«](#)
 - QOPT
2. The following file systems do not support timer values other than 0. An attempt to a value other than 0 for the timer value results in an `ERROR_TIMER_NOT_SUPPORTED` error.
 3. The following file systems do not support the atomic locking flag. If you turn on the atomic locking flag, an `ERROR_ATOMIC_LOCKS_NOT_SUPPORTED` error is returned.
 - Root
 - QOpenSys
 - User-Defined File System
 - QDLS
 - QOPT
 - QNetWare
 4. The following file systems do not support byte range locks. An attempt to use this API results in an `ERROR_GENERAL_FAILURE` error.
 - QSYS.LIB
 - [»Independent ASP QSYS.LIB«](#)
 - Network File System
 - QFileSvr.400 File System
 5. When you develop in C-based languages and this function is compiled with the `_LARGE_FILES` macro defined, it will be mapped to `DosSetFileLocks64()`. Additionally, the `PFILELOCK` data type will be mapped to a type `PFILELOCK64`.
 6. Locks placed on character special files result in advisory locks. For more information on advisory locking, please see the [fcntl\(\)--Perform File Control Command](#).

Related Information

- The `<fcntl.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<os2.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<os2def.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<bse.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<bsdos.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<bseerr.h>` file (see [Header Files for UNIX-Type Functions](#))
- [DosSetFileLocks64\(\)--Lock and Unlock a Byte Range of an Open File \(Large File Enabled\)](#)

Example

The following example opens, locks, and unlocks a file.

```
#define INCL_DOSERRORS
#define INCL_DOSFILEMGR
#include <os2.h>
```

```

#include <stdio.h>

#define NULL_RANGE 0L
#define LOCK_FLAGS 0

main() {
    char fn[]="lock.file";
    char buf[] =
        "Test data for locking and unlocking range of a file";
    int fd;
    ULONG lockTimeout = 2000; /* lock timeout of 2 seconds */
    FILELOCK Area;          /* area of file to lock/unlock */

    Area.Offset = 4;        /* start locking at byte 4 */
    Area.Range = 10;        /* lock 10 bytes for the file */

    /* Create a file for this example */
    fd = creat(fn, S_IWUSR | S_IRUSR);
    /* Write some data to the file */
    write(fd, buf, sizeof(buf) -1);
    close(fd);

    /* Open the file */
    if ((fd = open(fn, O_RDWR) < 0)
        {
            perror("open() error");
            return;
        }

    /* Lock a range */
    rc = DosSetFileLocks((HFILE)fd,
                        NULL_RANGE,
                        &Area,
                        &lockTimeout,
                        LOCK_FLAGS);
    if(rc != 0) /* Lock failed */
        {
            perror("DosSetFileLocks() error");
        }

    /* Unlock a range */
    rc = DosSetFileLocks((HFILE)fd,
                        &Area,
                        NULL_RANGE,
                        &lockTimeout,
                        LOCK_FLAGS);
    if(rc != 0) /* Unlock failed */
        {
            perror("DosSetFileLocks() error");
        }

    close(fd);
    unlink(fn);
}

```


DosSetFileLocks64()--Lock and Unlock a Byte Range of an Open File (Large File Enabled)

Syntax

```
#define INCL_DOSERRORS
#define INCL_DOSFILEMGR
#include <os2.h>

APIRET APIENTRY DosSetFileLocks64(HFILE FileHandle,
                                   PFILELOCK64 ppUnLockRange,
                                   PFILELOCK64 ppLockRange,
                                   ULONG ulTimeOut,
                                   ULONG ulFlags);
```

Threadsafe: Conditional; see Usage Notes.

The **DosSetFileLocks64()** function locks and unlocks a range of an open file. A non-zero range indicates that a lock or unlock request is being made.

The **DosSetFileLocks64()** treats the values specified in the **PFILELOCK64** structure as unsigned.

The maximum offset that can be specified using **DosSetFileLocks64()** is the largest value that can be held in an 8-byte, unsigned integer, $2^{64} - 1$.

The maximum length that can be specified using **DosSetFileLocks64()** is the largest value that can be held in an 8-byte, unsigned integer, $2^{64} - 1$.

DosSetFileLocks64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 byte as long as the file has been opened by either of the following:

- Using the **open64()** function (see [open64\(\)--Open File \(Large File Enabled\)](#)).
- Using the **open()** function (see [open\(\)--Open File](#)) with the **O_LARGEFILE** flag set in the **oflag** parameter. Note that the **PFILELOCK64** type will hold offsets greater than 2 GB minus 1 byte.

For details about parameters, authorities required, error conditions, and examples, see [DosSetFileLocks\(\)--Lock and Unlock a Byte Range of an Open File](#).

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **DosSetFileLocks64()** API and the **PFILELOCK64** data type, you must compile the source with **_LARGE_FILE_API** defined.
2. For additional usage notes about this API, see [Usage Notes](#) in the **DosSetFileLocks()** API.

Related Information

- The <fcntl.h> file (see [Header Files for UNIX-Type Functions](#))
- The <os2.h> file (see [Header Files for UNIX-Type Functions](#))
- The <os2def.h> file (see [Header Files for UNIX-Type Functions](#))
- The <bse.h> file (see [Header Files for UNIX-Type Functions](#))
- The <bsedos.h> file (see [Header Files for UNIX-Type Functions](#))
- The <bseerr.h> file (see [Header Files for UNIX-Type Functions](#))
- [DosSetFileLocks\(\)](#)--Lock and Unlock a Byte Range of an Open File

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

DosSetRelMaxFH()--Change Maximum Number of File Descriptors

Syntax

```
#define INCL_DOSERRORS
#define INCL_DOSFILEMGR
#include <os2.h>

APIRET APIENTRY DosSetRelMaxFH(PLONG pcbReqCount,
                                PULONG pcbCurMaxFH);
```

Threadsafe: Yes

The **DosSetRelMaxFH()** function requests that the system change the maximum number of file descriptors for the calling process (job). The system preserves all file descriptors that are currently open.

A request to increase the maximum number of file descriptors by more than the system can accommodate will succeed. The resulting maximum will be the largest number possible, but will be less than what you requested.

A request to decrease the maximum number of file descriptors will succeed. The resulting maximum will be the smallest number possible, but may be more than what you expected. For example, assume that the current maximum is 200 and there are 150 open files. A request to decrease the maximum by 75 results in the maximum being decreased by only 50, to 150, to preserve the open file descriptors.

A request to decrease the maximum number of file descriptors to below 20 will succeed, but the maximum will never be decreased below 20.

To retrieve the current maximum number of file descriptors, without any side effects, the value pointed to by *pcbReqCount* should be set to zero.

Parameters

pcbReqCount

(Input) A pointer to the number to be added to the maximum number of file descriptors for the calling process. If the value pointed to by *pcbReqCount* is positive, the system increases the maximum number of file descriptors. If the value pointed to by *pcbReqCount* is negative, the system decreases the maximum number of file descriptors.

pcbCurMaxFH

(Output) A pointer to the location to receive the new total number of allocated file descriptors.

Authorities

No authorization is required.

Return Value

NO_ERROR (0)

DosSetRelMaxFH() was successful. The function returns *NO_ERROR* (0) even if the system disregards or partially fulfills a request for an increase or a decrease (for example, decreasing by a smaller number than requested). You should examine the value pointed to by *pcbCurMaxFH* to determine the result of this function.

value

When *value* is not *NO_ERROR* (non-zero), **DosSetRelMaxFH()** was not successful. The *value* that is returned indicates the error.

Error Conditions

If **DosSetRelMaxFH()** is not successful, the value that is returned is one of the following errors. The `<bseerr.h>` header file defines these values.

[ERROR_GENERAL_FAILURE]

A general failure occurred.

This may result from damage in the system. Refer to messages in the job log for other possible causes.

[ERROR_PROTECTION_VIOLATION]

A protection violation occurred.

A pointer passed to this function is not a valid pointer.

Error Messages

The system may send the following messages from this function.

CPE3418 E

Possible APAR condition or hardware failure.

CPFA0D4 E

File system error occurred.

CPF3CF2 E

Error(s) occurred during running of &1 API.

CPF9872 E

Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. If you are using the `select()` API, you should be aware of the value of the `FD_SETSIZE` macro defined in the `<sys/types.h>` header file. This value is defined to be 200. This means that the `fd_set` structure is defined to contain 200 bits, one for each file descriptor.

If your application uses `DosSetRelMaxFH()` to increase the maximum number of file descriptors beyond 200, you should consider defining your own value for the `FD_SETSIZE` macro prior to including `<sys/types.h>`. This is to ensure that the `fd_set` structure is defined with the correct number of bits to accommodate the actual maximum number of file descriptors.

2. The maximum number of file descriptors for this process may be obtained by using the `sysconf()` API with the `_SC_OPEN_MAX` parameter.

Related Information

- The `<os2.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<os2def.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<bse.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<bosedos.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<bseerr.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<sys/types.h>` file (see [Header Files for UNIX-Type Functions](#))
- The [select\(\)](#) API
- [sysconf\(\)--Get System Configuration Variables](#)

Example

The following example increases the maximum number of file descriptors by two.

```
#define INCL_DOSERRORS
#define INCL_DOSFILEMGR
#include <os2.h>
#include <stdio.h>

void main()
{
    long   ReqCount = 0; /* Number to add to maximum */
                                /* file descriptor count. */
    ulong  CurMaxFH;    /* New count of file descriptors. */
    int    rc;         /* Return code. */

    /* Find out what the initial maximum is.*/
    if ( NO_ERROR == (rc = DosSetRelMaxFH(&ReqCount, &CurMaxFH)) )
    {
        printf("Initial maximum = %d",CurMaxFH);
    }
}
```

```
ReqCount = 2; /* Set up to increase by 2. */

if (NO_ERROR == (rc = DosSetRelMaxFH(&ReqCount, &CurMaxFH))
{
    printf("    New maximum = %d",CurMaxFH);
}
}
if (NO_ERROR != rc)
{
    printf("Error = &d",rc);
}
}
```

Output:

Initial maximum = 200 New maximum = 202

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

dup()--Duplicate Open File Descriptor

Syntax

```
#include <unistd.h>

int dup(int filde);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Yes

The **dup()** function returns a new open file descriptor. The new descriptor refers to the same open file as *filde* and shares any locks.

If the original file descriptor was opened in text mode, data conversion is also done on the duplicated file descriptor.

The FD_CLOEXEC flag that is associated with the new file descriptor is cleared. Refer to [fcntl\(\)--Perform File Control Command](#) for additional information about the FD_CLOEXEC flag.

Parameters

filde

(Input) A descriptor to be duplicated.

The following operations are equivalent:

```
fd = dup(filde);
fd = fcntl(filde, F_DUPFD, 0);
```

For further information, see [fcntl\(\)--Perform File Control Command](#).

Authorities

No authorization is required.

Return Value

value **dup()** was successful. The value returned is the new descriptor.

-1 **dup()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If `dup()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EBADF]

Descriptor not valid.

A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECANCEL]

Operation canceled.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[ENOSYS]

Function not implemented.

An attempt was made to use a function that is not available in this implementation for any object or any arguments.

The path name given refers to an object that does not support this function.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Related Information

- The <unistd.h> file (see [Header Files for UNIX-Type Functions](#))
- [close\(\)](#)--Close File or Socket Descriptor

- [creat\(\)](#)--Create or Rewrite File
- [dup2\(\)](#)--Duplicate Open File Descriptor to Another Descriptor
- [fcntl\(\)](#)--Perform File Control Command
- [open\(\)](#)--Open File

Example

The following example duplicates an open descriptor:

See [Code disclaimer information](#) for information pertaining to code examples.

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <errno.h>

void print_file_id(int file_descriptor) {
    struct stat info;
    if (fstat(file_descriptor, &info) != 0)
        fprintf(stderr, "stat() error for file_descriptor %d: %s\n",
                strerror(errno));
    else
        printf("The file id of file_descriptor %d is %d\n",
                file_descriptor, (int) info.st_ino);
}

main() {
    int file_descriptor, file_descriptor2;
    char fn[]="original.file";

    /* create original file */
    if((file_descriptor = creat(fn,S_IRUSR | S_IWUSR)) < 0)
        perror("creat() error");
    /* generate a duplicate file descriptor of file_descriptor */
    else {
        if ((file_descriptor2 = dup(file_descriptor)) < 0)
            perror("dup() error");
        /* print resulting information */
        else {
            print_file_id(file_descriptor);
            print_file_id(file_descriptor2);
            puts("The file descriptors are different but");
            puts("they point to the same file.");
            close(file_descriptor);
            close(file_descriptor2);
        }
    }
}
```



```
    unlink(fn);  
  }  
}
```

Output:

The file id of file_descriptor 0 is 30
The file id of file_descriptor 3 is 30
The file descriptors are different but
they point to the same file.

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

dup2()--Duplicate Open File Descriptor to Another Descriptor

Syntax

```
#include <unistd.h>

int dup2(int fildes, int fildes2);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see [Usage Notes](#).

The **dup2()** function returns a descriptor with the value *filde*s2. The descriptor refers to the same file as *filde*s, and it will close the file that *filde*s2 was associated with.

If the original file descriptor was opened in text mode, data conversion is also done on the duplicated file descriptor.

The FD_CLOEXEC flag that is associated with the new file descriptor is cleared. Refer to [fcntl\(\)](#)--Perform File Control Command for additional information about the FD_CLOEXEC flag.

The following conditions apply:

- If *filde*s2 is less than zero or greater than or equal to OPEN_MAX, **dup2()** returns -1 and sets the *errno* global variable to [EBADF].
- If *filde*s is a valid descriptor and is equal to *filde*s2, **dup2()** returns *filde*s2 without closing it.
- If *filde*s is not a valid descriptor, **dup2()** fails and does not close *filde*s2.

This function works with descriptors for any type of object.

Parameters

*filde*s

(Input) A descriptor to be duplicated.

*filde*s2

(Input) The descriptor to which the duplication is made.

Authorities

No authorization is required.

Return Value

value **dup2()** was successful. The value of *files2* is returned.

-1 **dup2()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **dup2()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EBADF]

Descriptor not valid.

A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - >>Independent ASP QSYS.LIB <<
 - QOPT

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))
- [close\(\)](#)--Close File or Socket Descriptor
- [creat\(\)](#)--Create or Rewrite File
- [dup\(\)](#)--Duplicate Open File Descriptor
- [fcntl\(\)](#)--Perform File Control Command
- [open\(\)](#)--Open File

Example

The following example duplicates an open descriptor:

See [Code disclaimer information](#) for information pertaining to code examples.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>

void print_file_id(int file_descriptor) {
    struct stat info;
    if (fstat(file_descriptor, &info) != 0)
        fprintf(stderr, "stat() error for file_descriptor %d: %s\n",
                strerror(errno));
    else
        printf("The file id of file_descriptor %d is %d\n", file_descriptor,
                (int) info.st_ino);
}

main() {
    int file_descriptor, file_descriptor2;
    char fn[] = "original.file";
    char fn2[] = "dup2.file";

    /* create original file */
    if((file_descriptor = creat(fn, S_IRUSR | S_IWUSR)) < 0)
        perror("creat() error");
    /* create file to dup to */
    else if((file_descriptor2 = creat(fn2, S_IWUSR)) < 0)
        perror("creat()error");
    /* dup file_descriptor to file_descriptor2; print results */
    else {
        print_file_id(file_descriptor);
    }
}
```

```

print_file_id(file_descriptor2);
if ((file_descriptor2 = dup2(file_descriptor, file_descriptor2)) < 0)
    perror("dup2() error");
else {
    puts("After dup2()...");
    print_file_id(file_descriptor);
    print_file_id(file_descriptor2);
    puts("The file descriptors are different but they");
    puts("point to the same file which is different than");
    puts("the file that the second file_descriptor originally pointed
to.");
    close(file_descriptor);
    close(file_descriptor2);
}
unlink(fn);
unlink(fn2);
}
}

```

Output:

```

The file id of file_descriptor 0 is 30
The file id of file_descriptor 3 is 58
After dup2()...
The file id of file_descriptor 0 is 30
The file id of file_descriptor 3 is 30
The file descriptors are different, but they
point to the same file, which is different than
the file that the second file_descriptor originally pointed to.

```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

» **accessx()--Determine File Accessibility for a Class of Users**

Syntax

```
#include <unistd.h>

int accessx(int filde, int amode, int who);
```

Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **accessx()** function determines whether a file can be accessed by a specified class of users in a particular manner.

Adopted authority is not used.

Parameters

filde

(Input) The file descriptor of the file that is having its accessibility checked.

amode

(Input) A bitwise representation of the access permissions to be checked.

The following symbols, which are defined in the **<unistd.h>** header file, can be used in *amode*:

F_OK

(x'00') Tests whether the file exists

R_OK

(x'04') Tests whether the file can be accessed for reading

W_OK

(x'02') Tests whether the file can be accessed for writing

X_OK

(x'01') Tests whether the file can be accessed for execution

You can take the bitwise inclusive OR of any or all of the last three symbols to test several access modes at once. If you are using *F_OK* to test for the existence of the file, you cannot use OR with any of the other symbols. If any other bits are set in *amode*, **accessx()** returns the [EINVAL] error.

who

(Input) The class of users whose authority is to be checked.

The following symbols, which are defined in the **<unistd.h>** header file, can be used in *who*:

ACC_SELF

(x'00') Determines if specified access is permitted for the current thread. The effective user and group IDs are used.

Note: If the real and effective user ID are the same and the real and effective group ID are the same, the request is treated as *ACC_INVOKER*. See the Usage Notes for more details.

ACC_INVOKER

(x'01') Determines if specified access is permitted for the current thread. The effective user and group IDs are used.

ACC_OTHERS

(x'08') Determines if specified access is permitted for any user other than the object owner. Only one of *R_OK*, *W_OK*, and *X_OK* is permitted when **who** is *ACC_OTHERS*. Privileged users (users with *ALLOBJ special authority) are not considered in this check.

ACC_ALL

(x'20') Determines if specified access is permitted for all users. Only one of *R_OK*, *W_OK*, and *X_OK* is permitted when **who** is *ACC_ALL*. Privileged users (users with *ALLOBJ special authority) are not considered in this check.

Authorities

The following authorities are required if the **who** parameter is *ACC_SELF* or *ACC_INVOKER*. If *ACC_SELF* is specified, the effective UID and GID of the caller are used. If *ACC_INVOKER* is used, the real UID and GID of the caller are used.

Authorization Required for `faccessx()`

Object Referred to	Authority Required	errno
Object when <i>R_OK</i> is specified	*R	EACCES
Object when <i>W_OK</i> is specified	*W	EACCES
Object when <i>X_OK</i> is specified	*X	EACCES
Object when <i>R_OK</i> <i>W_OK</i> is specified	*RW	EACCES
Object when <i>R_OK</i> <i>X_OK</i> is specified	*RX	EACCES
Object when <i>W_OK</i> <i>X_OK</i> is specified	*WX	EACCES
Object when <i>R_OK</i> <i>W_OK</i> <i>X_OK</i> is specified	*RWX	EACCES
Object when <i>F_OK</i> is specified	None	None

If the current thread has *ALLOBJ special authority, `faccessx()` with *ACC_SELF* or *ACC_INVOKER* will indicate success for *R_OK*, *W_OK*, or *X_OK* even if none of the permission bits are set.

Return Value

0

faccessx() was successful.

-1

faccessx() was not successful (or the specified access is not permitted for the class of users being checked). The *errno* global variable is set to indicate the error.

Error Conditions

If **faccessx()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

The class of users specified by the *who* parameter does not have the permission indicated by the *amode* parameter.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[EINTR]

Interrupted function call.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[ETXTBSY]

Text file busy.

An attempt was made to execute an OS/400 PASE program that is currently open for writing, or an attempt has been made to open for writing an OS/400 PASE program that is being executed.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E

Possible APAR condition or hardware failure.

CPFA0D4 E

File system error occurred. Error number &1.

CPF3CF2 E

Error(s) occurred during running of &1 API.

CPF9872 E

Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:
 - Where multiple threads exist in the job.
 - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - Independent ASP QSYS.LIB
 - QOPT
2. ACC_SELF Mapped to ACC_INVOKER

Some physical file systems do not support *ACC_SELF* for the *who* parameter. However, **facessx()** will

change the *who* parameter from *ACC_SELF* to *ACC_INVOKER* if the caller's real and effective user ID are equal, and the caller's real and effective group ID are equal.

3. Network File System Differences

The Network File System will only support the value *ACC_INVOKER* for the *who* parameter. If **faccessx()** is called on a file in a mounted Network File System directory with a value for *who* other than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP. **Note:** If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)

4. QNTC File System Differences

The QNTC File System will only support the value *ACC_INVOKER* for the *who* parameter. If **faccessx()** is called on a file in the QNTC File System with a value for *who* other than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP. **Note:** If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

5. QOPT File System Differences

If the file descriptor refers to an object that exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for the object follows the rules described in [the previous table](#), Authorization Required for **faccessx()**. If the object exists on a volume formatted in some other media format, no authorization checks are made on the object. The volume authorization list is checked for the requested authority regardless of the volume media format.

6. QFileSvr.400 File System Differences

The QFileSvr.400 File System will only support the value *ACC_INVOKER* for the *who* parameter. If **faccessx()** is called on a file in the QFileSvr.400 File System with a value for *who* other than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP. **Note:** If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

7. QNetWare File System Differences

The QNetWare File System will only support the value *ACC_INVOKER* for the *who* parameter. If **faccessx()** is called on a file in the QNetWare File System with a value for *who* other than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP. **Note:** If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<limits.h>` file (see [Header Files for UNIX-Type Functions](#))
- [chmod\(\)](#)--Change File Authorizations
- [open\(\)](#)--Open File
- [access\(\)](#)--Determine File Accessibility
- [accessx\(\)](#)--Determine File Accessibility for a Class of Users
- [QlgAccessx\(\)](#)--Determine File Accessibility for a Class of Users (using NLS-enabled path name)
- [QlgAccess\(\)](#)--Determine File Accessibility (using NLS-enabled path name)
- [stat\(\)](#)--Get File Information

Example

The following example determines how a file is accessed:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

main() {
    char path[]="/myfile";
    int fd;

    fd = open(path, O_RDONLY);
    if (fd == -1)
    {
        printf("Error opening file.\n");
        return;
    }

    if (faccessx(fd, R_OK, ACC_OTHERS) == 0)
        printf("Someone besides the owner has read access to '%s'\n", path);
    if (faccessx(fd, W_OK, ACC_OTHERS) == 0)
        printf("Someone besides the owner has write access to '%s'\n", path);
    if (faccessx(fd, X_OK, ACC_OTHERS) == 0)
        printf("Someone besides the owner has search access to '%s'\n", path);
    close(fd);
}
```

Output:

In this example **faaccessx()** was called on a descriptor for '/myfile'. The following would be the output if someone other than the owner has *R authority, someone besides the owner has *W authority, and noone other than the owner has *X authority.

```
Someone besides the owner has read access to '/'  
Someone besides the owner has write access to '/'
```



API introduced: V5R2

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

» fchdir()--Change Current Directory by Descriptor

Syntax

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

```
int fchdir(int fildev);
```

Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **fchdir()** function makes the directory named by *fildev* the new current directory. If the **fchdir()** function fails, the current directory is unchanged.

Parameters

fildev

(Input) The file descriptor of the directory.

Authorities

Note: Adopted authority is not used.

Authorization Required for fchdir()

Object Referred to	Authority Required	errno
Each directory of the path name	*X	EACCES

Return Value

0

fchdir() was successful.

-1

fchdir() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If `fhdir()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADF]

Descriptor not valid.

A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E

Possible APAR condition or hardware failure.

CPFA0D4 E

File system error occurred. Error number &1.

CPF3CF2 E

Error(s) occurred during running of &1 API.

CPF9872 E

Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - Independent ASP QSYS.LIB
 - QOPT

The **fchdir()** API operates on two objects: the previous current working directory and the new one. If either of these objects is managed by a file system that is not threadsafe, **fchdir()** fails with the ENOTSAFE error code.

2. Network File System Differences

If the local storage of attributes and names is not suppressed (option noac when the file system is mounted), then one can potentially use the **fchdir()** API to change to a directory which has been removed. This depends on how often and when the local storage of attributes and names is refreshed.

Related Information

- The <**unistd.h**> file (see [Header Files for UNIX-Type Functions](#))
- The <**limits.h**> file (see [Header Files for UNIX-Type Functions](#))
- [chdir\(\)--Change Current Directory](#)
- [getcwd\(\)--Get Current Directory](#)

- [QlgChdir\(\)--Change Current Directory](#)
- [QlgGetcwd\(\)--Get Current Directory](#)

Example

The following example uses **fchdir()**:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

main() {
    char dir[]="tempfile";
    int file_descriptor;
    int oflag1 = O_RDONLY | O_CCSID;
    mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR;
    unsigned int open_ccsid = 37;

    if ((file_descriptor = open(dir,oflag1,mode,open_ccsid)) < 0)
        perror("open() error");
    else {
        if (fchdir(file_descriptor) != 0)
            perror("fchdir() to tempfile failed");
        close(file_descriptor);
    }
}
```

Output:

```
fchdir() to tempfile failed: Not a directory.
```



API introduced: V5R2

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

fchmod()--Change File Authorizations by Descriptor

Syntax

```
#include <sys/stat.h>

int fchmod(int fildev, mode_t mode);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **fchmod()** function changes S_ISUID, S_ISGID, and the permission bits of the open file or directory identified by *fildev* its file descriptor, to the corresponding bits specified in *mode*. **fchmod()** has no effect on file descriptions for files that are open at the time **fchmod()** is called.

fchmod() marks for update the change time of the file.

If the file is checked out by another user (someone other than the user profile of the current job), **fchmod()** fails with the [EBUSY] error.

Parameters

fildev

(Input) The file descriptor of the file.

mode

(Input) Bits that define S_ISUID, S_ISGID, and the access permissions of the file.

The *mode* argument is created with one of the symbols defined in the <sys/stat.h> header file. For more information on the symbols, refer to [chmod\(\)--Change File Authorizations](#).

If bits other than the bits listed above are set in *mode*, **fchmod()** returns the [EINVAL] error.

Authorities

Note: Adopted authority is not used.

Figure 1-14. Authorization Required for fchmod() (excluding QDLS)

Object Referred to	Authority Required	errno
Object	Owner (see Note)	EPERM

Note: You do not need the listed authority if you have *ALLOBJ special authority.

Figure 1-15. Authorization Required for fchmod() in the QDLS File System

Object Referred to	Authority Required	errno
Object	Owner or *ALL	EACCES

Return Value

0

fchmod() was successful.

-1

fchmod() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **fchmod()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADF]

Descriptor not valid.

A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

➤ *[EJRNDAMAGE]*

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNENTTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.

[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal.❧

[ENAMETOOLONG]

A path name is too long.

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

❧*[ENEWJRN]*

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal.❧

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOSYS]

Function not implemented.

An attempt was made to use a function that is not available in this implementation for any object or any arguments.

The path name given refers to an object that does not support this function.

[ENOSYSRSC]

System resources not available to complete request.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

The object referenced by the descriptor does not support the function.

[EPERM]

Operation not permitted.

You must have appropriate privileges or be the owner of the object or other resource to do the

requested operation.

The thread does not have authority to perform the requested function.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this API:

CPE3418 E

Possible APAR condition or hardware failure.

CPFA0D4 E

File system error occurred. Error number &1.

CPF3CF2 E

Error(s) occurred during running of &1 API.

CPF9872 E

Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. [➤](#) All of the usage notes for **chmod()** apply to **fchmod()**. See [Usage Notes](#) in the **chmod** API. [⏪](#)

Related Information

- The `<sys/stat.h>` file (see [Header Files for UNIX-Type Functions](#))
- [chmod\(\)--Change File Authorizations](#)
- [chown\(\)--Change Owner and Group of File](#)
- [fchown\(\)--Change Owner and Group of File by Descriptor](#)
- [mkdir\(\)--Make Directory](#)
- [open\(\)--Open File](#)
- [stat\(\)--Get File Information](#)

Example

The following example changes a file permission:

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
```

```

main() {
    char fn[]="temp.file";
    int file_descriptor;
    struct stat info;

    if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        if (stat(fn, &info)!= 0)
            perror("stat() error");
        else {
            printf("original permissions were: %08o\n", info.st_mode);
        }
        if (fchmod(file_descriptor, S_IRWXU|S_IRWXG) != 0)
            perror("fchmod() error");
        else {
            if (stat(fn, &info)!= 0)
                perror("stat() error");
            else {
                printf("after fchmod(), permissions are: %08o\n", info.st_mode);
            }
        }
        if (close(file_descriptor)!= 0)
            perror("close() error");
        if (unlink(fn)!= 0)
            perror("unlink() error");
    }
}

```

Output:

```

original permissions were: 00100200
after fchmod(), permissions are: 00100770

```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

fchown()--Change Owner and Group of File by Descriptor

Syntax

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

```
int fchown(int fildev, uid_t owner, gid_t group);
```

Threadsafe: Conditional; see Usage Notes.

The **fchown()** function changes the owner and group of a file. The permissions of the previous owner or primary group to the object are revoked.

If the file is checked out by another user (someone other than the user profile of the current job), **fchown()** fails with the [EBUSY] error.

When **fchown()** completes successfully, it marks the change time of the file to be updated.

Parameters

fildev

(Input) The file descriptor of the file.

owner

(Input) The new user ID to be set for file.

group

(Input) The new group ID to be set for file.

Note: Changing the owner or the primary group causes the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode to be cleared, unless the caller has *ALLOBJ special authority. If the caller does have *ALLOBJ special authority, the bits are not changed. This does not apply to directories, FIFO special files, or pipes. See the [chmod\(\)](#) documentation.

Authorities

Note: Adopted authority is not used.

Figure 1-16. Authorization Required for fchown() (excluding QSYS.LIB, independent ASP QSYS.LIB, and QDLS)

Object Referred to	Authority Required	errno
Object, when changing the owner	Owner and *OBJEXIST (also see Note 1)	EPERM
Object, when changing the primary group	See Note 2	EPERM

Previous owner's user profile, when changing the owner	*DLT	EPERM
New owner's user profile, when changing the owner	*ADD	EPERM
User profile of previous primary group, when changing the primary group	*DLT	EPERM
New primary group's user profile, when changing the primary group	*ADD	EPERM
Note:		
<ol style="list-style-type: none"> 1. You do not need the listed authority if you have *ALLOBJ special authority. 2. At least one of the following must be true: <ol style="list-style-type: none"> a. You have *ALLOBJ special authority. b. You are the owner <u>and</u> either of the following: <ul style="list-style-type: none"> ■ The new primary group is the primary group of the job. ■ The new primary group is one of the supplementary groups of the job. 		

Figure 1-17. Authorization Required for fchown() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Object, when changing the owner	See Note (1)	EPERM
Object, when changing the primary group	See Note (2)	EPERM
Note: The required authorization varies for each object type. See the following commands in the iSeries Security Reference book for details:		
<ol style="list-style-type: none"> 1. CHGOBJOWN 2. CHGOBJPGP 		

Figure 1-18. Authorization Required for fchown() in the QDLS File System

Object Referred to	Authority Required	errno
Object	*ALLOBJ Special Authority or Owner	EPERM
Previous owner's user profile, when changing the owner	*DLT	EPERM
New owner's user profile, when changing the owner	*ADD	EPERM
Previous primary group's user profile, when changing the primary group	*DLT	EPERM
New primary group's user profile, when changing the primary group	*ADD	EPERM

Figure 1-19. Authorization Required for fchown() in the QOPT File System

Object Referred to	Authority Required	errno
Volume authorization list	*CHANGE	EACCES
Each directory in the path name preceding the object.	*X	EACCES
Object	*ALLOBJ Special Authority or Owner	EPERM

Return Value

0

fchown() was successful.

-1

fchown() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **fchown()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADF]

Descriptor not valid.

A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL. arameter passed to this function is not valid.

owner or *group* is not a valid user ID (uid) or group ID (gid).

owner is the current primary group of the object.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

➤ *[EJRNDAMAGE]*

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNTTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.

[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal. <<

[ENAMETOOLONG]

A path name is too long.

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

>>*[ENEWJRN]*

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal. <<

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOSYS]

Function not implemented.

An attempt was made to use a function that is not available in this implementation for any object or any arguments.

The path name given refers to an object that does not support this function.

[ENOSYSRSC]

System resources not available to complete request.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

The object referenced by the descriptor does not support the function.

[EPERM]

Operation not permitted.

You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.

The thread does not have authority to perform the requested function.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.



The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

Error Messages

The following messages may be sent from this function:

CPE3418 E Possible APAR condition or hardware failure.
CPFA0D4 E File system error occurred. Error number &1.
CPF3CF2 E Error(s) occurred during running of &1 API.
CPF9872 E Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - Independent ASP QSYS.LIB 
 - QOPT

2. QDLS File System Differences

The owner and primary group of the /QDLS directory (root folder) cannot be changed. If an attempt is made to change the owner and primary group, a [ENOTSUP] error is returned.

3. QOPT File System Differences

Changing the owner and primary group is allowed only for an object that exists on a volume

formatted in Universal Disk Format (UDF). For all other media formats, ENOTSUP will be returned.

QOPT file system objects that have owners will not be recognized by the Work with Objects by Owner (WRKOBJOWN) CL command. Likewise, QOPT objects that have a primary group will not be recognized by the Work Objects by Primary Group (WRKOBJPGP) CL command.

4. QFileSvr.400 File System Differences

The QFileSvr.400 file system does not support **fchown()**.

5. QNetWare File System Differences

Primary group is not supported. The GID must be zero on this API.

6. QNTC File System Differences

The owner of files and directories cannot be changed. All files and directories in QNTC are owned by the QDFTOWN user profile.

Related Information

- The <**unistd.h**> file (see [Header Files for UNIX-Type Functions](#))
- [chown\(\)--Change Owner and Group of File](#)
- [chmod\(\)--Change File Authorizations](#)
- [fchmod\(\)--Change File Authorizations by Descriptor](#)
- [mkdir\(\)--Make Directory](#)
- [open\(\)--Open File](#)
- [stat\(\)--Get File Information](#)

Example

The following example changes the owner ID and group ID:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

main() {
    char fn[]="temp.file";
    int file_descriptor;
    struct stat info;

    if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
```

```
    perror("creat() error");
else {
    stat(fn, &info);
    printf("original owner was %d and group was %d\n", info.st_uid,
           info.st_gid);
    if (fchown(file_descriptor, 152, 0) != 0)
        perror("fchown() error");
    else {
        stat(fn, &info);
        printf("after fchown(), owner is %d and group is %d\n",
               info.st_uid, info.st_gid);
    }
    close(file_descriptor);
    unlink(fn);
}
}
```

Output:

```
original owner was 137 and group was 0
after fchown(), owner is 152 and group is 0
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

fcntl()--Perform File Control Command

Syntax

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

int fcntl(int descriptor,
          int command,
          ...)
```

Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see [Usage Notes](#).

The **fcntl()** function performs various actions on open descriptors, such as obtaining or changing the attributes of a file or socket descriptor.

Parameters

descriptor

(Input) The descriptor on which the control command is to be performed, such as having its attributes retrieved or changed.

command

(Input) The command that is to be performed on the *descriptor*.

...

(Input) A variable number of optional parameters that is dependent on the *command*. Only some of the commands use this parameter.

The *fcntl()* commands that are supported are:

F_DUPFD Duplicates the descriptor. A third **int** argument must be specified. **fcntl()** returns the lowest descriptor greater than or equal to this third argument that is not already associated with an open file. This descriptor refers to the same object as *descriptor* and shares any locks. If the original descriptor was opened in text mode, data conversion is also done on the duplicated descriptor. The **FD_CLOEXEC** flag that is associated with the new descriptor is cleared.

F_GETFD Obtains the descriptor flags for *descriptor*. **fcntl()** returns these flags as its result. For a list of supported file descriptor flags, see [Flags](#). Descriptor flags are associated with a single descriptor and do not affect other descriptors that refer to the same object.

- F_GETFL* Obtains the file status flags and file access mode flags for *descriptor*. **fcntl()** returns these flags as its result. For a list of supported file status and file access mode flags, see [Using the oflag Parameter in open\(\)](#).
- F_GETLK* Obtains locking information for an object. You must specify a third argument of type struct flock *. See [File Locking](#) for details. **fcntl()** returns 0 if it successfully obtains the locking information. When you develop in C-based languages and the function is compiled with the `_LARGE_FILES` macro defined, `F_GETLK` is mapped to the `F_GETLK64` symbol.
- F_GETLK64* Obtains locking information for a large file. You must specify a third argument of type struct flock64 *. See [File Locking](#) for details. **fcntl()** returns 0 if it successfully obtains the locking information. When you develop in C-based languages, it is necessary to compile the function with the `_LARGE_FILE_API` macro defined to use this symbol.
- F_GETOWN* Returns the process ID or process group ID that is set to receive the SIGIO (I/O is possible on a descriptor) and SIGURG (urgent condition is present) signals. For more information, see [Signal APIs](#).
- F_SETFD* Sets the descriptor flags for *descriptor*. You must specify a third *int* argument, which gives the new file descriptor flag settings (see [Flags](#)). If any other bits in the third argument are set, **fcntl()** fails with the [EINVAL] error. **fcntl()** returns 0 if it successfully sets the flags. Descriptor flags are associated with a single descriptor and do not affect other descriptors that refer to the same object.
- F_SETFL* Sets status flags for the descriptor. You must specify a third *int* argument, giving the new file status flag settings (see [Flags](#)). **fcntl()** does not change the file access mode, and file access bits in the third argument are ignored. All other oflag values that are valid on the **open()** API are also ignored. If any other bits in the third argument are set, **fcntl()** fails with the [EINVAL] error. **fcntl()** returns 0 if it successfully sets the flags.
- F_SETLK* Sets or clears a file segment lock. You must specify a third argument of type struct flock *. See [File Locking](#) for details. **fcntl()** returns 0 if it successfully clears the lock. When you develop in C-based languages and the function is compiled with the `_LARGE_FILES` macro defined, `F_SETLK` is mapped to the `F_SETLK64` symbol.
- F_SETLK64* Sets or clears a file segment lock for a large file. You must specify a third argument of type struct flock64 *. See [File Locking](#) for details. **fcntl()** returns 0 if it successfully clears the lock. When you develop in C-based languages, it is necessary to compile the function with the `_LARGE_FILE_API` macro defined to use this symbol.
- F_SETLKW* Sets or clears a file segment lock; however, if a shared or exclusive lock is blocked by other locks, **fcntl()** waits until the request can be satisfied. You must specify a third argument of type struct flock *. See [File Locking](#) for details. When you develop in C-based languages and the function is compiled with the `_LARGE_FILES` macro defined, `F_SETLKW` is mapped to the `F_SETLKW64` symbol.

- F_SETLKW64* Sets or clears a file segment lock on a large file; however, if a shared or exclusive lock is blocked by other locks, **fcntl()** waits until the request can be satisfied. See [File Locking](#) for details. You must specify a third argument of type `struct flock64 *`. When you develop in C-based languages, it is necessary to compile the function with the `_LARGE_FILE_API` macro defined to use this symbol.
- F_SETOWN* Sets the process ID or process group ID that is to receive the SIGIO and SIGURG signals. For more information, see [Signal APIs](#).

Flags

There are several types of flags associated with each open object. Flags for an object are represented by symbols defined in the `<fcntl.h>` header file. The following *file status* flags can be associated with an object:

- FASYNC* The SIGIO signal is sent to the process when it is possible to do I/O.
- FNDELAY* This flag is defined to be equivalent to `O_NDELAY`.
- O_APPEND* Append mode. If this flag is 1, every write operation on the file begins at the end of the file.
- O_DSYNC* Synchronous update - data only. If this flag is 1, all file data is written to permanent storage before the update operation returns. Update operations include, but are not limited to, the following: **ftruncate()**, **open()** with `O_TRUNC`, and **write()**.
- O_NDELAY* This flag is defined to be equivalent to `O_NONBLOCK`.
- O_NONBLOCK* Non-blocking mode. If this flag is 1, read or write operations on the file will not cause the thread to block. This file status flag applies only to pipe, FIFO, and socket descriptors.
- *O_RSYNC* Synchronous read. If this flag is 1, read operations to the file will be performed synchronously. This flag is used in combination with `O_SYNC` or `O_DSYNC`. When `O_RSYNC` and `O_SYNC` are set, all file data and file attributes are written to permanent storage before the read operation returns. When `O_RSYNC` and `O_DSYNC` are set, all file data is written to permanent storage before the read operation returns.⏪
- *O_SYNC* Synchronous update. If this flag is 1, all file data and file attributes relative to the I/O operation are written to permanent storage before the update operation returns. Update operations include, but are not limited to, the following: **ftruncate()**, **open()** with `O_TRUNC`, and **write()**.⏪

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.

A mask can be used to extract flags:

`O_ACCMODE` Extracts file access mode flags.

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

`FD_CLOEXEC` Controls descriptor inheritance during `spawn()` and `spawnp()` when simple inheritance is being used, as follows:

- If the `FD_CLOEXEC` flag is zero, the descriptor is inherited by the child process that is created by the `spawn()` or `spawnp()` API.
Note: Descriptors that are created as a result of the `opendir()` API (to implement open directory streams) are not inherited, regardless of the value of the `FD_CLOEXEC` flag.
- If the `FD_CLOEXEC` flag is set, the descriptor is not inherited by the child process that is created by the `spawn()` or `spawnp()` API.

Refer to [spawn\(\)](#)--Spawn Process and [spawnp\(\)](#)--Spawn Process with Path for additional information about `FD_CLOEXEC`.

File Locking

A local or remote job can use `fcntl()` to lock out other local or remote jobs from a part of a file. By locking out other jobs, the job can read or write to that part of the file without interference from others. File locking can ensure data integrity when several jobs have a file accessed concurrently. For more information about remote locking, see information about the network lock manager and the network status monitor in the

[OS/400 Network File System Support](#)  book.

Two different structures are used to control locking operations: `struct flock` and `struct flock64` (both defined in the `<fcntl.h>` header file). You can use `struct flock64` with the `F_GETLK64`, `F_SETLK64`, and `F_SETLKW64` commands to control locks on large files (files greater than 2GB minus 1 byte). The `struct flock` structure has the following members:

short	<code>l_type</code>	Indicates the type of lock, as indicated by one of the following symbols (defined in the <code><fcntl.h></code> header file): <ul style="list-style-type: none"> <code>F_RDLCK</code> Indicates a <i>read lock</i>; also called a <i>shared lock</i>. When a job has a read lock, no other job can obtain write locks for that part of the file. More than one job can have a read lock on the same part of a file simultaneously. To establish a read lock, a job must have the file accessed for reading. <code>F_WRLCK</code> Indicates a <i>write lock</i>; also called an <i>exclusive lock</i>. When a job has a write lock, no other job can obtain a read lock or write lock on the same part or an overlapping part of that file. A job cannot put a write lock on part of a file if another job already has a read lock on an overlapping part of the file. To establish a write lock, a job must have accessed the file for writing. <code>F_UNLCK</code> Unlocks a lock that was set previously.
short	<code>l_whence</code>	One of three symbols used in determining the part of the file that is affected by this lock. These symbols are defined in the <code><unistd.h></code> header file and are the same as symbols used by <code>lseek()</code> : <ul style="list-style-type: none"> <code>SEEK_CUR</code> The current file offset in the file. <code>SEEK_END</code> The end of the file. <code>SEEK_SET</code> The start of the file.
off_t	<code>l_start</code>	Gives a byte offset used to identify the part of the file that is affected by this lock. If <code>l_start</code> is negative, it is handled as an unsigned value. The part of the file affected by the lock begins at this offset from the location given by <code>l_whence</code> . For example, if <code>l_whence</code> is <code>SEEK_SET</code> and <code>l_start</code> is 10, the locked part of the file begins at an offset of 10 bytes from the beginning of the file.
off_t	<code>l_len</code>	Gives the size of the locked part of the file, in bytes. If the size is negative, it is treated as an unsigned value. If <code>l_len</code> is zero, the locked part of the file begins at the position specified by <code>l_whence</code> and <code>l_start</code> , and extends to the end of the file. Together, <code>l_whence</code> , <code>l_start</code> , and <code>l_len</code> are used to describe the part of the file that is affected by this lock.
pid_t	<code>l_pid</code>	Specifies the job ID of the job that holds the lock. This is an output field used only with <code>F_GETLK</code> actions.
void	<code>*l_reserved0</code>	Reserved. Must be set to NULL.
void	<code>*l_reserved1</code>	Reserved. Must be set to NULL.

When you develop in C-based languages and this function is compiled with `_LARGE_FILES` defined, the struct `flock` data type will be mapped to a struct `flock64` data type. To use the struct `flock64` data type explicitly, it is necessary to compile the function with `_LARGE_FILE_API` defined.

The struct `flock64` structure has the following members:

short	l_type	Indicates the type of lock, as indicated by one of the following symbols (defined in the <code><fcntl.h></code> header file): <ul style="list-style-type: none"> <code>F_RDLCK</code> Indicates a <i>read lock</i>; also called a <i>shared lock</i>. When a job has a read lock, no other job can obtain write locks for that part of the file. More than one job can have a read lock on the same part of a file simultaneously. To establish a read lock, a job must have the file accessed for reading. <code>F_WRLCK</code> Indicates a <i>write lock</i>; also called an <i>exclusive lock</i>. When a job has a write lock, no other job can obtain a read lock or write lock on the same part or an overlapping part of that file. A job cannot put a write lock on part of a file if another job already has a read lock on an overlapping part of the file. To establish a write lock, a job must have accessed the file for writing. <code>F_UNLCK</code> Unlocks a lock that was set previously.
short	l_whence	One of three symbols used in determining the part of the file that is affected by this lock. These symbols are defined in the <code><unistd.h></code> header file and are the same as symbols used by <code>lseek()</code> : <ul style="list-style-type: none"> <code>SEEK_CUR</code> The current file offset in the file. <code>SEEK_END</code> The end of the file. <code>SEEK_SET</code> The start of the file.
char	l_reserved2[4]	Reserved field
off64_t	l_start	Gives a byte offset used to identify the part of the file that is affected by this lock. <code>l_start</code> is handled as a signed value. The part of the file affected by the lock begins at this offset from the location given by <code>l_whence</code> . For example, if <code>l_whence</code> is <code>SEEK_SET</code> and <code>l_start</code> is 10, the locked part of the file begins at an offset of 10 bytes from the beginning of the file.
off64_t	l_len	Gives the size of the locked part of the file, in bytes. If the size is negative, the part of the file affected is <code>l_start + l_len</code> through <code>l_start - 1</code> . If <code>l_len</code> is zero, the locked part of the file begins at the position specified by <code>l_whence</code> and <code>l_start</code> , and extends to the end of the file. Together, <code>l_whence</code> , <code>l_start</code> , and <code>l_len</code> are used to describe the part of the file that is affected by this lock.
pid_t	l_pid	Specifies the job ID of the job that holds the lock. This is an output field used only with <code>F_GETLK</code> actions.
char	reserved3[4]	Reserved field.
void	*l_reserved0	Reserved. Must be set to NULL.
void	*l_reserved1	Reserved. Must be set to NULL.

You can set locks by specifying `F_SETLK` or `F_SETLK64` as the *command* argument for `fcntl()`. Such a function call requires a third argument pointing to a struct flock structure (or struct flock64 in the case of `F_SETLK64`), 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(file_descriptor, F_SETLK, &lock_it);

```

This example sets up a 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` and `F_SETLKW64` operations are similar to `F_SETLK` and `F_SETLK64`, except that they wait until the lock can be set. For example, if you want to establish an exclusive lock and some other job already has a lock established on an overlapping part of the file, **fcntl()** waits until the other process has removed its lock.

`F_SETLKW` and `F_SETLKW64` operations can encounter *deadlocks* when job A is waiting for job B to unlock a region and job B is waiting for job A to unlock a different region. If the system detects that an `F_SETLKW` or `F_SETLKW64` might cause a deadlock, **fcntl()** fails with *errno* set to `[EDEADLK]`.

With the `F_SETLK64`, `F_SETLKW64`, and `F_GETLK64` operations, the maximum offset that can be specified is the largest value that can be held in an 8-byte, signed integer.

A job can determine locking information about a file by using `F_GETLK` and `F_GETLK64` as the *command* argument for **fcntl()**. In this case, the call to **fcntl()** should specify a third argument pointing to a 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 job ID of the job 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.

If **fcntl()** attempts to operate on a large file (one larger than 2GB minus 1 byte) with the `F_SETLK`, `F_GETLK`, or `FSETLKW` commands, the API fails with `[E_OVERFLOW]`. To work with large files, compile with the `_LARGE_FILE_API` macro defined (when you develop in C-based languages) and use the `F_SETLK64`, `F_GETLK64`, or `FSETLKW64` commands. When you develop in C-based languages, it is also possible to work with large files by compiling the source with the `_LARGE_FILES` macro label defined. Note that the file must have been opened for large file access (either the **open64()** API was used or the **open()** API was used with the `O_LARGEFILE` flag defined in the `oflag` parameter).

An application that uses the `F_SETLK` or `F_SETLKW` commands may try to lock or unlock a file that has been extended beyond 2GB minus 1 byte by another application. If the value of `l_len` is set to 0 on the lock or unlock request, the byte range held or released will go to the end of the file rather than ending at offset 2GB minus 2.

An application that uses the `F_SETLK` or `F_SETLKW` commands also may try to lock or unlock a file that has been extended beyond offset 2GB minus 2 with `l_len` NOT set to 0. If this application attempts to lock or unlock the byte range up to offset 2GB minus 2 and `l_len` is not 0, the unlock request will unlock the file only up to offset 2GB minus 2 rather than to the end of the file.

A job can have several locks on a file at the same time, but only one type of lock can be set on a given byte.

Therefore, if a job puts a new lock on a part of a file that it had locked previously, the job 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.

Locks can start and extend beyond the current end of a file, but cannot start or extend ahead of the beginning of a file.

All of the locks a job has on a file are removed when the job closes any descriptor that refers to the locked file.

All locks obtained using **fcntl()** are advisory only. Jobs can use advisory locks to inform each other that they want to protect parts of a file, but advisory locks do not prevent input and output on the locked parts. If a job has appropriate permissions on a file, it can perform whatever I/O it chooses, regardless of what advisory locks are set. Therefore, advisory locking is only a convention, and it works only when all jobs respect the convention.

Another type of lock, called a mandatory lock, can be set by a remote personal computer application. Mandatory locks restrict I/O on the locked parts. A read fails when reading a part that is locked with a mandatory write lock. A write fails when writing a part that is locked with a mandatory read or mandatory write lock.

The maximum starting offset that can be specified by using the **fcntl()** API is $2^{63} - 1$, the largest number that can be represented by a signed 8-byte integer. Mandatory locks set by a personal computer application or by a user of the **DosSetFileLocks64()** API may lock a byte range that is greater than $2^{63} - 1$.

An application that uses the **F_SETLK64** or **F_SETLKW64** commands can lock the offset range that is beyond $2^{63} - 1$ by locking offset $2^{63} - 1$. When offset $2^{63} - 1$ is locked, it implicitly locks to the end of the file. The end of the file is the largest number that can be represented by an 8-byte unsigned integer or $2^{64} - 1$. This implicit lock may inhibit the personal computer application from setting mandatory locks in the range not explicitly accessible by the **fcntl()** API.

Any lock set using the **fcntl()** API that locks offset $2^{63} - 1$ will have a length of 0.

An application that uses the **F_GETLK64** may encounter a mandatory lock set by a personal computer application, which locks a range of offsets greater than $2^{63} - 1$. This lock conflict will have a starting offset equal to or less than $2^{63} - 1$ and a length of 0.

Authorities

No authorization is required.

Return Value

value **fcntl()** was successful. The value returned depends on the *command* that was specified.

-1 **fcntl()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If `fcntl()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

- [EACCES]** Permission denied.
- An attempt was made to access an object in a way forbidden by its object access permissions.
- The thread does not have access to the specified file, directory, component, or path.
- If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
- [EAGAIN]** Operation would have caused the process to be suspended.
- The process tried to lock with `F_SETLK`, but the lock is in conflict with a previously established lock.
- [EBADF]** Descriptor not valid.
- A descriptor argument was out of range, referred to an object that was not open, or a read or write request was made to an object that is not open for that operation.
- A given descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open object.
- [EBADFID]** A file ID could not be assigned when linking an object to a directory.
- The file ID table is missing or damaged.
- To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.
- [EBADFUNC]** Function parameter in the signal function is not set.
- A given descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open object.
- [EBUSY]** Resource busy.
- An attempt was made to use a system resource that is not available at this time.
- [EDAMAGE]** A damaged object was encountered.
- A referenced object is damaged. The object cannot be used.
- [EDEADLK]** Resource deadlock avoided.
- An attempt was made to lock a system resource that would have resulted in a deadlock situation. The lock was not obtained.
- The function attempted was failed to prevent a deadlock.

- [EFAULT]* The address used for an argument is not correct.
- In attempting to use an argument in a call, the system detected an address that is not valid.
- While attempting to access a parameter passed to this function, the system detected an address that is not valid.
- [EINVAL]* The value specified for the argument is not correct.
- A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.
- An argument value is not valid, out of range, or NULL.
- [EIO]* Input/output error.
- A physical I/O error occurred.
- A referenced object may be damaged.
- [EMFILE]* Too many open files for this process.
- An attempt was made to open more files than allowed by the value of OPEN_MAX. The value of OPEN_MAX can be retrieved using the sysconf() function.
- The process has more than OPEN_MAX descriptors already open (see the **sysconf()** function).
- [ENOLCK]* No locks available.
- A system-imposed limit on the number of simultaneous file and record locks was reached, and no more were available at that time.
- [ENOMEM]* Storage allocation request failed.
- A function needed to allocate storage, but no storage is available.
- There is not enough memory to perform the requested function.
- [ENOSYS]* Function not implemented.
- An attempt was made to use a function that is not available in this implementation for any object or any arguments.
- The path name given refers to an object that does not support this function.
- [ENOTAVAIL]* Independent Auxiliary Storage Pool (ASP) is not available.
- The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.
- To recover from this error, wait until processing has completed for the independent ASP.
- [ENOTSAFE]* Function is not allowed in a job that is running with multiple threads.

[E_OVERFLOW] Object is too large to process.

The object's data size exceeds the limit allowed by this function.

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

The command argument is `F_GETLK`, `F_SETLK`, or `F_SETLKW` and the offset of any byte in the requested segment cannot be represented correctly in a variable of type `off_t` (the offset is greater than 2GB minus 1 byte).

[E_STALE] File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[E_UNKNOWN] Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

[EADDRNOTAVAIL] Address not available.

[ECONNABORTED] Connection ended abnormally.

[ECONNREFUSED] The destination socket refused an attempted connect operation.

[ECONNRESET] A connection with a remote socket was reset by that socket.

[EHOSTDOWN] A remote host is not available.

[EHOSTUNREACH] A route to the remote host is not available.

[ENETDOWN] The network is not currently available.

[ENETRESET] A socket is connected to a host that is no longer available.

[ENETUNREACH] Cannot reach the destination network.

[ETIMEDOUT] A remote host did not respond within the timeout period.

[EUNATCH] The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPFA0D4 E	File system error occurred. Error number &1.
CPFA081 E	Unable to set return value or error code.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - [»Independent ASP QSYS.LIB «](#)
 - QOPT

2. [QSYS.LIB »](#)and [Independent ASP QSYS.LIB «](#)File System Differences


The following **fcntl()** commands are not supported:

- F_GETLK
- F_SETLK
- F_SETLKW

Using any of these commands results in an [ENOSYS] error.

3. Network File System Differences

Reading and writing to a file with the Network File System relies on byte-range locking to guarantee data integrity. To prevent data inconsistency, use the **fcntl()** API to get and release these

locks. For more information about remote locking, see information about the network lock manager and the network status monitor in the [OS/400 Network File System Support](#)  book.

4. QNetWare File System Differences


F_GETLK and F_SETLKW are not supported. F_RDLCK and F_WRLCK are ignored. All locks prevent reading and writing. Advisory locks are not supported. All locks are mandatory locks. Locking a file that is opened more than once in the same job with the same access mode is not supported, and its result is undefined.

5. This function will fail with the [E_OVERFLOW] error if the command is F_GETLK, F_SETLK, or F_SETLKW and the offset or the length exceeds offset 2 GB minus 2.

6. When you develop in C-based languages and an application is compiled with the _LARGE_FILES macro defined, the struct flock data type will be mapped to a struct flock64 data type. To use the struct flock64 data type explicitly, it is necessary to compile the function with the _LARGE_FILE_API defined.

7. In several cases, similar function can be obtained by using *ioctl()*.

Related Information

- The <sys/types.h> file (see [Header Files for UNIX-Type Functions](#))
- The <unistd.h> file (see [Header Files for UNIX-Type Functions](#))
- The <fcntl.h> file (see [Header Files for UNIX-Type Functions](#))
- [close\(\)](#)--Close File or Socket Descriptor
- [dup\(\)](#)--Duplicate Open File Descriptor
- [dup2\(\)](#)--Duplicate Open File Descriptor to Another Descriptor
- [ioctl\(\)](#)--Perform I/O Control Request
- [lseek\(\)](#)--Set File Read/Write Offset
- [open\(\)](#)--Open File
- [spawn\(\)](#)--Spawn Process
- [spawnp\(\)](#)--Spawn Process with Path
- [OS/400 Network File System Support](#)  book

Example

The following example uses **fcntl()**:

See [Code disclaimer information](#) for information pertaining to code examples.

```
#include <stdio.h>
#include <sys/types.h>
```

```

#include <unistd.h>
#include <fcntl.h>

int main()
{
    int flags;
    int append_flag;
    int nonblock_flag;
    int access_mode;
    int file_descriptor; /* File Descriptor */
    char *text1 = "abcdefghij";
    char *text2 = "0123456789";
    char read_buffer[25];

    memset(read_buffer, '\0', 25);

    /* create a new file */
    file_descriptor = creat("testfile",S_IRWXU);
    write(file_descriptor, text1, 10);
    close(file_descriptor);

    /* open the file with read/write access */
    file_descriptor = open("testfile", O_RDWR);
    read(file_descriptor, read_buffer,24);
    printf("first read is \'%s\'\n",read_buffer);

    /* reset file pointer to the beginning of the file */
    lseek(file_descriptor, 0, SEEK_SET);
    /* set append flag to prevent overwriting existing text */
    fcntl(file_descriptor, F_SETFL, O_APPEND);
    write(file_descriptor, text2, 10);
    lseek(file_descriptor, 0, SEEK_SET);
    read(file_descriptor, read_buffer,24);
    printf("second read is \'%s\'\n",read_buffer);

    close(file_descriptor);
    unlink("testfile");

    return 0;
}

```

Output:

```

first read is 'abcdefghij'
second read is 'abcdefghij0123456789'

```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

fpathconf()--Get Configurable Path Name Variables by Descriptor

Syntax

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

```
long fpathconf(int file_descriptor, int name);
```

Threadsafe: Conditional; see Usage Notes.

The **fpathconf()** function determines the value of a configuration variable (*name*) associated with a particular file descriptor (*file_descriptor*). **fpathconf()** works exactly like **pathconf()**, except that it takes a file descriptor as an argument rather than taking a path name.

Parameters

file_descriptor

(Input) A file descriptor of the file for which the value of the configurable variable is requested.

name

(Input) The name of the configuration variable value requested.

The value of *name* can be any one of a set of symbols defined in the `<unistd.h>` include file. Each symbol stands for a configuration variable. The possible symbols are as follows:

`_PC_CHOWN_RESTRICTED`

Represents `_POSIX_CHOWN_RESTRICTED`, as defined in the `<unistd.h>` header file. It restricts use of **chown()** to a job with appropriate privileges, and allows the group ID of a file to be changed only to the effective group ID of the job or to one of its supplementary group IDs. If *file_descriptor* is a directory, **fpathconf()** returns the value for any kind of file under the directory, but not for subdirectories of the directory.

`_PC_LINK_MAX`

Represents `LINK_MAX`, which indicates the maximum number of links the file can have. If *file_descriptor* is a directory, **pathconf()** returns the maximum number of links that can be established to the directory itself.

`_PC_MAX_CANON`

Represents `MAX_CANON`, which indicates the maximum number of bytes in a terminal canonical input line.

`_PC_MAX_INPUT`

Represents `MAX_INPUT`, which indicates the minimum number of bytes for which space is available in a terminal input queue. This available space is the maximum number of bytes that a portable application can have the user enter before the application actually reads the input.

`_PC_NAME_MAX`

Represents `NAME_MAX`, which indicates the maximum number of characters in a file name (not including any terminating null at the end if the file name is stored as a string). This symbol refers

only to the file name itself; that is, the last component of the path name of the file. **fpathconf()** returns the maximum length of file names, even when the path does not refer to a directory.

_PC_PATH_MAX

Represents `PATH_MAX`, which indicates the maximum number of characters in a complete path name (not including any terminating null at the end if the path name is stored as a string).

fpathconf() returns the maximum length of a path name relative to the root of the file system that is managing the object indicated by *file_descriptor*, even when the path does not refer to a directory.

_PC_PIPE_BUF

Represents `PIPE_BUF`, which indicates the maximum number of bytes that can be written "atomically" to a pipe. If more than this number of bytes are written to a pipe, the operation may take more than one physical write operation and physical read operation to read the data on the other end of the pipe. If *file_descriptor* is a FIFO special file, **fpathconf()** returns the value for the file itself. If *file_descriptor* is a directory, **fpathconf()** returns the value for any FIFOs that exist or that can be created under the directory. If *file_descriptor* is any other kind of file, an error of `[EINVAL]` is returned.

_PC_NO_TRUNC

Represents `_POSIX_NO_TRUNC`, as defined in the `<unistd.h>` header file. It generates an error if a file name is longer than `NAME_MAX`. If *file_descriptor* refers to a directory, the value returned by **fpathconf()** applies to all files under that directory.

_PC_VDISABLE

Represents `_POSIX_VDISABLE`, as defined in the `<unistd.h>` header file. This symbol indicates that terminal special characters can be disabled using this character value, if it is defined.

_PC_THREAD_SAFE

This symbol is used to determine if the object represented by *path* resides in a threadsafe file system. **fpathconf()** returns the value 1 if the file system is threadsafe and 0 if the file system is not threadsafe. **fpathconf()** will never fail with error code `[ENOTSAFE]` when called with `_PC_THREAD_SAFE`.

If *file_descriptor* is a descriptor for a socket, **fpathconf()** returns an error of `[EINVAL]`.

Authorities

No authorization is required.

Return Value

value

fpathconf() was successful. The value of the variable requested in *name* is returned.

-1

One of the following has occurred:

- A particular variable has no limit (for example, `_PC_PATH_MAX`). The *errno* global variable is not changed.
- **fpathconf()** was not successful. The *errno* is set.

Error Conditions

If **fpathconf()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADF]

Descriptor not valid.

A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL. *name* is not a valid configuration variable name, or the given variable cannot be associated with the specified file.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC

- QSYS.LIB
- [»Independent ASP QSYS.LIB«](#)
- QOPT

Related Information

- The <unistd.h> file (see [Header Files for UNIX-Type Functions](#))
- [open\(\)--Open File](#)
- [pathconf\(\)--Get Configurable Path Name Variables](#)
- [»QlPathconf\(\)--Get Configurable Path Name Variables«](#)

Example

The following example uses **fpathconf()**:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

main() {
    long result;
    char fn[]="temp.file";
    int file_descriptor;

    if ((file_descriptor = creat(fn, S_IRUSR)) < 0)
        perror("creat() error");
    else {
        errno = 0;
        puts("examining NAME_MAX limit for current working directory's");
        puts("filesystem:");
        if ((result = fpathconf(file_descriptor, _PC_NAME_MAX)) == -1)
            if (errno == 0)
                puts("There is no limit to NAME_MAX.");
            else perror("fpathconf() error");
        else
            printf("NAME_MAX is %ld\n", result);
        close(file_descriptor);
        unlink(fn);
    }
}
```

Output:

```
examining NAME_MAX limit for current working directory's
filesystem:
NAME_MAX is 255
```

API introduced: V5R2

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

fstat()--Get File Information by Descriptor

Syntax

```
#include <sys/stat.h>

int fstat(int descriptor,
          struct stat *buffer)
```

Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see [Usage Notes](#).

The **fstat()** function gets status information about the object specified by the open descriptor *descriptor* and stores the information in the area of memory indicated by the *buffer* argument. The status information is returned in a stat structure, as defined in the `<sys/stat.h>` header file.

Parameters

descriptor

(Input) The descriptor for which information is to be retrieved.

buffer

(Output) A pointer to a buffer of type **struct stat** in which the information is returned. The structure pointed to by the *buffer* parameter is described in [stat\(\)-- Get File Information](#).

The *st_mode*, *st_dev*, and *st_blksize* fields are the only fields set for socket descriptors. The *st_mode* field is set to a value that indicates the descriptor is a socket descriptor, the *st_dev* field is set to -1, and the *st_blksize* field is set to an optimal value determined by the system.

Authorities

No authorization is required.

Return Value

0 **fstat()** was successful. The information is returned in *buffer*.

-1 **fstat()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **fstat()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

- [EACCES]** Permission denied.
- An attempt was made to access an object in a way forbidden by its object access permissions.
- The thread does not have access to the specified file, directory, component, or path.
- If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
- [EAGAIN]** Operation would have caused the process to be suspended.
- [EBADF]** Descriptor not valid.
- A descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.
- A given descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open object.
- [EBADFID]** A file ID could not be assigned when linking an object to a directory.
- The file ID table is missing or damaged.
- To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.
- [EBADFUNC]** Function parameter in the signal function is not set.
- A given descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open object.
- [EBUSY]** Resource busy.
- An attempt was made to use a system resource that is not available at this time.
- [EDAMAGE]** A damaged object was encountered.
- A referenced object is damaged. The object cannot be used.

- [EFAULT]** The address used for an argument is not correct.
- In attempting to use an argument in a call, the system detected an address that is not valid.
- While attempting to access a parameter passed to this function, the system detected an address that is not valid. [EFAULT] is returned if this function is passed a pointer parameter that is not valid.
- [EINVAL]** The value specified for the argument is not correct.
- A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.
- An argument value is not valid, out of range, or NULL.
- This error code may be returned when the underlying object represented by the descriptor is unable to fill the **stat** structure (for example, if the function was issued against a socket descriptor that had its connection reset).
- [EIO]** Input/output error.
- A physical I/O error occurred.
- A referenced object may be damaged.
- [ENOBUFFS]** There is not enough buffer space for the requested operation.
- [ENOSYSRSC]** System resources not available to complete request.
- [ENOTAVAIL]** Independent Auxiliary Storage Pool (ASP) is not available.
- The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.
- To recover from this error, wait until processing has completed for the independent ASP.
- [ENOTSAFE]** Function is not allowed in a job that is running with multiple threads.
- [EOVERFLOW]** Object is too large to process.
- The object's data size exceeds the limit allowed by this function.
- The specified file exists and its size is too large to be represented in the structure pointed to by *buffer* (the file is larger than 2GB minus 1 byte).
- [EPERM]** Operation not permitted.
- You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.
- [ESTALE]** File or object handle rejected by server.
- If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNATCH] The protocol required to support the specified address family is not available at this time.

[EUNKNOWN] Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

[EADDRNOTAVAIL] Address not available.

[ECONNABORTED] Connection ended abnormally.

[ECONNREFUSED] The destination socket refused an attempted connect operation.

[ECONNRESET] A connection with a remote socket was reset by that socket.

[EHOSTDOWN] A remote host is not available.

[EHOSTUNREACH] A route to the remote host is not available.

[ENETDOWN] The network is not currently available.

[ENETRESET] A socket is connected to a host that is no longer available.

[ENETUNREACH] Cannot reach the destination network.

[ETIMEDOUT] A remote host did not respond within the timeout period.



Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPFA0D4 E	File system error occurred. Error number &1.
CPFA081 E	Unable to set return value or error code.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPE3418 E	Possible APAR condition or hardware failure.

CPF9872 E Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:
 - Where multiple threads exist in the job.
 - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - Independent ASP QSYS.LIB 
 - QOPT

2. Sockets-Specific Notes

- The field *st_mode* can be inspected using the S_ISSOCK macro (defined in <sys/stat.h>) to determine if the descriptor is pointing to a socket descriptor.
- For socket descriptors, use the send buffer size (this is the value returned for *st_blksize*) for the length parameter on your input and output functions. This can improve performance.

Note: IBM reserves the right to change the calculation of the optimal send size.

3. QOPT File System Differences

The value for *st_atime* will always be zero. The value for *st_ctime* will always be the creation date and time of the file or directory.

The user, group, and other mode bits are always on for an object that exists on a volume not formatted in Universal Disk Format (UDF).

fstat on /QOPT will always return 2,147,483,647 for size fields.

fstat on optical volumes will return the volume capacity or 2,147,483,647, whichever is smaller.

The file access time is not changed.

4. Network File System Differences

Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values

may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)

5. QNetWare File System Differences

The QNetWare file system does not fully support mode bits. See the [Netware on iSeries](#) topic for more information.

6. This function will fail with the [E_OVERFLOW] error if the specified file exists and its size is too large to be represented in the structure pointed to by *buffer* (the file is larger than 2GB minus 1 byte).
7. When you develop in C-based languages and this function is compiled with `_LARGE_FILES` defined, it will be mapped to `fstat64()`. Note that the type of the *buffer* parameter, `struct stat *`, also will be mapped to type `struct stat64 *`. See [stat64\(\)](#) for more information on this structure.

Related Information

- The `<sys/types.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<sys/stat.h>` file (see [Header Files for UNIX-Type Functions](#))
- [fcntl\(\)](#)--Perform File Control Command
- [fstat64\(\)](#)--Get File Information by Descriptor (Large File Enabled)
- [lstat\(\)](#)--Get File or Link Information
- [open\(\)](#)--Open File
- [socket\(\)](#)--Create Socket
- [stat\(\)](#)--Get File Information
- [stat64\(\)](#)--Get File Information (Large File Enabled)

Example

See [Code disclaimer information](#) for information pertaining to code examples.

The following example gets status information:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <time.h>

main() {
    char fn[]="temp.file";
    struct stat info;
    int file_descriptor;

    if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
```

```

else {
    if (fstat(file_descriptor, &info) != 0)
        perror("fstat() error");
    else {
        puts("fstat() returned:");
        printf("  inode:   %d\n",    (int) info.st_ino);
        printf(" dev id:   %d\n",    (int) info.st_dev);
        printf("  mode:    %08x\n",    info.st_mode);
        printf("  links:   %d\n",        info.st_nlink);
        printf("   uid:    %d\n",    (int) info.st_uid);
        printf("   gid:    %d\n",    (int) info.st_gid);
    }
    close(file_descriptor);
    unlink(fn);
}
}
}

```

Output: Note that the output may vary from system to system.

```

fstat() returned:
inode:   3057
dev id:   1
mode:    03000080
links:   1
uid:     137
gid:     500

```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

fstat64()--Get File Information by Descriptor (Large File Enabled)

Syntax

```
#include <sys/stat.h>

int fstat64(int fildev, struct stat64 *buf);
```

Threadsafe: Conditional; see Usage Notes.

The **fstat64()** function gets status information about the file specified by the open file descriptor *file_descriptor* and stores the information in the area of memory indicated by the *buf* argument. The status information is returned in a stat64 structure, as defined in the `<sys/stat.h>` header file.

fstat64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 byte as long as the file has been opened by either of the following:

- Using the **open64()** function (see [open64\(\)--Open File \(Large File Enabled\)](#)).
- Using the **open()** function (see [open\(\)--Open File](#)) with `O_LARGEFILE` set in the *oflag* parameter.

The elements of the stat64 structure are described in [stat64\(\)--Get File Information \(Large File Enabled\)](#).

For additional information about parameters, authorities required, and error conditions, see [fstat\(\)--Get File Information by Descriptor](#).

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **fstat64()** API and the struct stat64 data type, you must compile the source with the `_LARGE_FILE_API` macro defined.
2. All of the usage notes for **fstat()** apply to **fstat64()**. See [Usage Notes](#) in the **fstat()** API.

Example

The following example gets status information:

```
#define _LARGE_FILE_API
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <time.h>

main() {
    char fn[]="temp.file";
```

```

struct stat64 info;
int file_descriptor;

if ((file_descriptor = creat64(fn, S_IWUSR)) < 0)
    perror("creat64() error");
else {
    if (ftruncate64(file_descriptor, 8589934662) != 0)
        perror("ftruncate64() error");
    else {
        if (fstat64(file_descriptor, &info) != 0)
            perror("fstat64() error");
        else {
            puts("fstat64() returned:");
            printf("  inode:   %d\n",    (int) info.st_ino);
            printf(" dev id:   %d\n",    (int) info.st_dev);
            printf("  mode:   %08x\n",    info.st_mode);
            printf("  links:  %d\n",    info.st_nlink);
            printf("   uid:   %d\n",    (int) info.st_uid);
            printf("   gid:   %d\n",    (int) info.st_gid);
            printf("   size:  %lld\n", (long long) info.st_size);
        }
    }
    close(file_descriptor);
    unlink(fn);
}
}
}

```

Output: Note that the output may vary from system to system.

```

fstat64() returned:
  inode:   3057
 dev id:   1
  mode:   03000080
  links:  1
   uid:   137
   gid:   500
   size:  8589934662

```

fstatvfs()--Get File System Information by Descriptor

Syntax

```
#include <sys/statvfs.h>

int fstatvfs(int fildev, struct statvfs *buf);
```

Threadsafe: Conditional; see Usage Notes.

The **fstatvfs()** function gets status information about the file system that contains the file referenced by the open file descriptor *fildev*. The information is stored in the area of memory indicated by the *buf* argument. The status information is returned in a statvfs structure, as defined in the `<sys/statvfs.h>` header file.

Parameters

fildev

(Input) The file descriptor of the file from which file system information is required.

buf

(Output) A pointer to the area to which the information should be written.

The elements of the statvfs structure are described in [statvfs\(\)--Get File System Information](#). Signed fields of the statvfs structure that are not supported by the mounted file system will be set to -1.

Authorities

Note: Adopted authority is not used.

Figure 1-20. Authorization Required for fstatvfs()

Object Referred to	Authority Required	errno
Each directory in the path name that precedes the object	*X	EACCES
Object	None	None

Return Value

0

fstatvfs() was successful. The information is returned in *buf*.

-1

fstatvfs() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If `fstatvfs()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADF]

Descriptor not valid.

A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than `POSIX_SYMLINK` (defined in the `limits.h` header file). Symbolic links are encountered during resolution of the directory or path name.

[ENAMETOOLONG]

A path name is too long.

A path name is longer than `PATH_MAX` characters or some component of the name is longer than

NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using the **pathconf()** function.

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[EPERM]

Operation not permitted.

You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E

Possible APAR condition or hardware failure.

CPFA0D4 E

File system error occurred. Error number &1.



CPF3CF2 E

Error(s) occurred during running of &1 API.

CPF9872 E

Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - Independent ASP QSYS.LIB 
 - QOPT

2. Root ("/") and QOpenSys File System Differences

These file systems return the `f_flag` field with the `ST_NOSUID` flag bit turned off. However, support for the `setuid/setgid` semantics is limited to the ability to store and retrieve the `S_ISUID` and `S_ISGID` flags when these file systems are accessed from the Network File System server.

3. Network File System Differences

Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)

4. When you develop in C-based languages and an application is compiled with the `_LARGE_FILES` macro defined, the `fstatvfs()` API will be mapped to a call to the `fstatvfs64()`. Additionally, the struct `statvfs` data type will be mapped to a struct `statvfs64`.

Related Information

- The `<sys/statvfs.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<sys/types.h>` file (see [Header Files for UNIX-Type Functions](#))
- [chmod\(\)--Change File Authorizations](#)
- [chown\(\)--Change Owner and Group of File](#)
- [creat\(\)--Create or Rewrite File](#)
- [dup\(\)--Duplicate Open File Descriptor](#)
- [fcntl\(\)--Perform File Control Command](#)
- [fstatvfs64\(\)--Get File System Information by Descriptor \(64-Bit Enabled\)](#)
- [link\(\)--Create Link to File](#)
- [open\(\)--Open File](#)
- [read\(\)--Read from Descriptor](#)
- [statvfs\(\)--Get File System Information](#)
- [unlink\(\)--Remove Link to File](#)
- [utime\(\)--Set File Access and Modification Times](#)
- [write\(\)--Write to Descriptor](#)

Example

The following example gets status information about a file system:

```
#include <sys/statvfs.h>
#include <stdio.h>

main() {
    struct statvfs info;
    int fildes;

    if (-1 == (fildes = open("/",O_RDONLY)))
        perror("open() error");
    else if (-1 == fstatvfs(fildes, &info))
        perror("fstatvfs() error");
    else {
        puts("fstatvfs() returned the following information");
        puts("about the Root ('/') file system:");
        printf("  f_bsize      : %u\n", info.f_bsize);
        printf("  f_blocks     : %08X%08X\n",
```

```

        *((int *)&info.f_blocks[0]),
        *((int *)&info.f_blocks[4]));
printf("  f_bfree      : %08X%08X\n",
        *((int *)&info.f_bfree[0]),
        *((int *)&info.f_bfree[4]));
printf("  f_files       : %u\n", info.f_files);
printf("  f_ffree        : %u\n", info.f_ffree);
printf("  f_fsid         : %u\n", info.f_fsid);
printf("  f_flag         : %X\n", info.f_flag);
printf("  f_namemax      : %u\n", info.f_namemax);
printf("  f_pathmax      : %u\n", info.f_pathmax);
printf("  f_basetype     : %s\n", info.f_basetype);
}
}

```

Output: The following information will vary from file system to file system.

statvfs() returned the following information about the Root ('/') file system:

```

f_bsize      : 4096
f_blocks     : 00000000002BF800
f_bfree      : 0000000000091703
f_files      : 4294967295
f_ffree      : 4294967295
f_fsid       : 0
f_flag       : 1A
f_namemax    : 255
f_pathmax    : 4294967295
f_basetype   : "root" (/)

```

API introduced: V4R2

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

fstatvfs64()--Get File System Information by Descriptor (64-Bit Enabled)

Syntax

```
#include <sys/statvfs.h>

int fstatvfs64(int fildev, struct statvfs64 *buf);
```

Threadsafe: Conditional; see Usage Notes.

The **fstatvfs64()** function gets status information about the file system that contains the file referred to by the open file descriptor *fildev*. The information is stored in the area of memory indicated by the *buf* argument. The status information is returned in a `statvfs64` structure, as defined in the `<sys/statvfs.h>` header file.

For details about parameters, authorities required, error conditions and examples, see [fstatvfs\(\)--Get File System Information by Descriptor](#). For details about the struct `statvfs64` structure, see [statvfs64\(\)--Get File System Information \(64-Bit Enabled\)](#).

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **fstatvfs64()** API and the struct `statvfs64` data type, you must compile the source with the `_LARGE_FILE_API` defined.
2. All of the usage notes for **fstatvfs()** apply to **fstatvfs64()**. See [Usage Notes](#) in the **fstatvfs()** API.

fsync()--Synchronize Changes to File

Syntax

```
#include <unistd.h>

int fsync(int file_descriptor);
```

Threadsafe: Conditional; see Usage Notes.

The **fsync()** function transfers all data for the file indicated by the open file descriptor *file_descriptor* to the storage device associated with *file_descriptor*. **fsync()** does not return until the transfer is complete, or until an error is detected.

Parameters

file_descriptor

(Input) The file descriptor of the file that is to have its modified data written to permanent storage.

Authorities

No authorization is required. Authorization is verified during **open()** or **creat()**.

Return Value

0

fsync() was successful.

-1

fsync() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **fsync()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[*EACCES*]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file

permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADF]

Descriptor not valid.

A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

The file type is not valid for this operation.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

➤ *[EJRNDAMAGE]*

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.

[EJRMENTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.

[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal.

[ENEWJRN]

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal. <<

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages



The following messages may be sent from this function:

CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code *[ENOTSAFE]* when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe.

Only the following file systems are threadsafe for this function:

- Root
- QOpenSys
- User-defined
- QNTC
- QSYS.LIB
- Independent ASP QSYS.LIB 
- QOPT

2. Using this function on a character special file will result in a return value of -1 and the errno global value set to EINVAL.

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))
- [open\(\)--Open File](#)
- [write\(\)--Write to Descriptor](#)

Example

The following example uses `fsync()`:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

#define mega_string_len 250000

main() {
    char *mega_string;
    int file_descriptor;
    int ret;
    char fn[]="fsync.file";

    if ((mega_string = (char*) malloc(mega_string_len)) == NULL)
        perror("malloc() error");
    else if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        memset(mega_string, 's', mega_string_len);
        if ((ret = write(file_descriptor,
                        mega_string, mega_string_len)) == -1)
            perror("write() error");
        else {
```

```
    printf("write() wrote %d bytes\n", ret);
    if (fsync(file_descriptor) != 0)
        perror("fsync() error");
    else if ((ret = write(file_descriptor,
                          mega_string, mega_string_len)) == -1)
        perror("write() error");
    else
        printf("write() wrote %d bytes\n", ret);
}
close(file_descriptor);
unlink(fn);
}
}
```

Output:

```
write() wrote 250000 bytes
write() wrote 250000 bytes
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)



ftruncate()--Truncate File

Syntax

```
#include <unistd.h>

int ftruncate(int file_descriptor, off_t length);
```

Threadsafe: Conditional; see Usage Notes.

The **ftruncate()** function truncates the file indicated by the open file descriptor *file_descriptor* to the indicated *length*. *file_descriptor* must be a "regular file" that is open for writing. (A regular file is a stream file that can support positioning the file offset.) If the file size exceeds *length*, any extra data is discarded. If the file size is smaller than *length*, the file is extended and filled with binary zeros to the indicated length. (In the QSYS.LIB  and independent ASP QSYS.LIB file systems  blanks are used instead of zeros to pad records after a member is extended.)

If **ftruncate()** completes successfully, it marks the change time and modification times of the file. Also, the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode are cleared. If **ftruncate()** is not successful, the file is unchanged.

If **ftruncate()** is used to truncate the file to 0 bytes and the file has an OS/400 digital signature, the signature is deleted.

Parameters

file_descriptor

(Input) The file descriptor of the file.

length

(Input) The desired size of the file in bytes.

Authorities

No authorization is required. Authorization is verified during **open()** or **creat()**.

Return Value

0

ftruncate() was successful.

-1

ftruncate() was not successful. The *errno* global variable is set to indicate the error. If the file descriptor is not open for writing, **ftruncate** returns a [EBADF] error. If the file descriptor is a valid descriptor open for writing but is not a descriptor for a regular file, **ftruncate()** returns a

[EINVAL] error.

Error Conditions

If **ftruncate()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADF]

Descriptor not valid.

A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

The QSYS.LIB >> or independent ASP QSYS.LIB <<< file system cannot get exclusive access to the member to clear truncated data.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.



[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFBIG]

Object is too large.

The size of the object would exceed the system allowed maximum size  or the process soft file size limit. 

The file is a regular file and *length* is greater than 2GB minus 1 byte.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL. *file_descriptor* does not refer to a regular file open for writing, or the specified length is not correct.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[EISDIR]

Specified target is a directory.

The path specified named a directory where a file or object name was expected.

The path name given is a directory.

 *[EJRNDDAMAGE]*

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNENTTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.

[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal. <<

[ELOCKED]

Area being read from or written to is locked.

The read or write of an area conflicts with a lock held by another process.

[ENAMETOOLONG]

A path name is too long.

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

>> *[ENEWJRN]*

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal. <<

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOSYS]

Function not implemented.

An attempt was made to use a function that is not available in this implementation for any object or any arguments.

The path name given refers to an object that does not support this function.

[ENOSYSRSC]

System resources not available to complete request.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

The object referenced by the descriptor does not support the function.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E Possible APAR condition or hardware failure.
CPFA0D4 E File system error occurred. Error number &1.
CPF3CF2 E Error(s) occurred during running of &1 API.
CPF9872 E Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - >>Independent ASP QSYS.LIB <<
 - QOPT
2. When you develop in C-based languages and this function is compiled with `_LARGE_FILES` defined, it will be mapped to `ftruncate64()`. Note also that the type of the *length* parameter will be remapped from `off_t` to `off64_t`.
3. For the Network File System, this function will fail with the [EFBIG] or the [EIO] error if the length specified is greater than the largest file size supported by the server.
4. Using this function on a character special file results in a return value of -1 and the `errno` global value set to `EINVAL`.

5. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

This function is not supported for save files and will fail with error code [ENOTSUP].

6. If the write exceeds the process soft file size limit, signal SIFXFSZ is issued.

Related Information

- The <[unistd.h](#)> file (see [Header Files for UNIX-Type Functions](#))
- [ftruncate64\(\)--Truncate File \(Large File Enabled\)](#)
- [open\(\)--Open File](#)

Example

The following example uses **ftruncate()**:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

#define string_len 1000

main() {
    char *mega_string;
    int file_descriptor;
    int ret;
    char fn[]="write.file";
    struct stat st;

    if ((mega_string = (char*) malloc(string_len)) == NULL)
        perror("malloc() error");
    else if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        memset(mega_string, '0', string_len);
        if ((ret = write(file_descriptor, mega_string, string_len)) == -1)
            perror("write() error");
        else {
            printf("write() wrote %d bytes\n", ret);
            fstat(file_descriptor, &st);
            printf("the file has %ld bytes\n", (long) st.st_size);
            if (ftruncate(file_descriptor, 1) != 0)
                perror("ftruncate() error");
            else {
                fstat(file_descriptor, &st);
                printf("the file has %ld bytes\n", (long) st.st_size);
            }
        }
    }
}
```

```
    }
    close(file_descriptor);
    unlink(fn);
}
}
```

Output:

```
write() wrote 1000 bytes
the file has 1000 bytes
the file has 1 bytes
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

ftruncate64()--Truncate File (Large File Enabled)

Syntax

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

```
int ftruncate64(int file_descriptor, off64_t length);
```

Threadsafe: Conditional; see Usage Notes.

The **ftruncate64()** function truncates the file indicated by the open file descriptor *file_descriptor* to the indicated *length*. *file_descriptor* must be a "regular file" that is open for writing. (A regular file is a stream file that can support positioning the file offset.) If the file size exceeds *length*, any extra data is discarded. If the file size is smaller than *length*, the file is extended and filled with binary zeros to the indicated length. (In the QSYS.LIB [Q](#) and independent ASP QSYS.LIB file systems, [Q](#) blanks are used instead of zeros to pad records after a member is extended.)

ftruncate64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 byte as long as the file has been opened by either of the following:

- Using the **open64()** function (see [open64\(\)--Open File \(Large File Enabled\)](#)).
- Using the **open()** function (see [open\(\)--Open File](#)) with the O_LARGEFILE flag set in the oflag parameter.

If **ftruncate64()** completes successfully, it marks the change time and modification times of the file. If **ftruncate64()** is not successful, the file is unchanged.

For additional information about parameters, authorities, error conditions, and examples, see [ftruncate\(\)--Truncate File](#).

Usage Notes

1. For file systems that do support large files, this function will fail with the [EFBIG] error if the *length* specified is greater than 2GB minus 1 byte and O_LARGEFILE is not set in the oflag.
2. For file systems that do not support large files, this function will fail with the [EINVAL] error if the *length* specified is greater than 2GB minus 1 byte.
3. QFileSvr.400 File System Differences

Although QFileSvr.400 does not support large files, it will return [EFBIG] if the *length* specified is greater than 2GB minus 1 byte.

4. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **ftruncate64()** API and the off64_t data type, you must compile the source with _LARGE_FILE_API defined.
5. All of the usage notes for **ftruncate()** apply to **ftruncate64()**. See [Usage Notes](#) in the **ftruncate()** API.

API introduced: V4R4

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

getcwd()--Get Current Directory

Syntax

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

Threadsafe: Conditional; see Usage Notes.

The **getcwd()** function determines the absolute path name of the current directory and stores it in *buf*. The components of the returned path name are not symbolic links.

The access time of each directory in the absolute path name of the current directory (excluding the current directory itself) is updated.

If *buf* is a NULL pointer, **getcwd()** returns a NULL pointer and the [EINVAL] error.

Parameters

buf

(Output) A pointer to a buffer that will be used to hold the absolute path name of the current directory. The buffer must be large enough to contain the full pathname including the terminating NULL character. The current directory is returned in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See [QlgGetcwd\(\)--Get Current Directory](#) for a description and an example of supplying the *buf* in any CCSID.

size

(Input) The number of bytes in the buffer *buf*.

Authorities

Note: Adopted authority is not used.

Authorization Required for getcwd()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the current directory	*RX	EACCES
Current directory	*X	EACCES

Note: QDLS File System Differences

If the current directory is an immediate subdirectory of /QDLS (that is, at the next level below /QDLS in the directory hierarchy), the user must have *RX (*USE) authority to the directory. Otherwise, the QDLS authority requirements are the same as shown above.

Return Value

value

getcwd() was successful. The value returned is a pointer to *buf*.

NULL

getcwd() was not successful. The *errno* global variable is set to indicate the error. After an error, the contents of *buf* are not defined.

Note: If *buf* is a *NULL* pointer, **getcwd()** returns a *NULL* pointer.

Error Conditions

If **getcwd()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[*EACCES*]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[*EAGAIN*]

Operation would have caused the process to be suspended.

[*EBADFID*]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[*EBADNAME*]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EEXIST]

File exists.

The file specified already exists and the specified operation requires that it not exist.

The named file, directory, or path already exists.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

A parameter passed to this function is not valid.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[EMFILE]

Too many open files for this process.

An attempt was made to open more files than allowed by the value of OPEN_MAX. The value of OPEN_MAX can be retrieved using the sysconf() function.

The process has more than OPEN_MAX descriptors already open (see the **sysconf()** function).

[ENAMETOOLONG]

A path name is too long.

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

[ENFILE]

Too many open files in the system.

A system limit has been reached for the number of files that are allowed to be concurrently open in the system.

The entire system has too many other file descriptors already open.

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[ERANGE]

A range error occurred.

The value of an argument is too small, or a result too large.

The **size** argument is too small. It is greater than zero but smaller than the length of the path name plus a NULL character.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code *[ENOTSAFE]* when both of the following conditions occur:
 - Where multiple threads exist in the job.
 - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys

- User-defined
- QNTC
- QSYS.LIB
- [»Independent ASP QSYS.LIB«](#)
- QOPT

2. QOPT File System Differences

If the directory exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for the directory and preceding directories in the path name follows the rules described in [Authorization Required for getcwd\(\)](#). If the directory exists on a volume formatted in some other media format, no authorization checks are made on the directory or preceding directories. The volume authorization list is checked for *USE authority regardless of the volume media format.

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))
- [chdir\(\)--Change Current Directory](#)
- [QlgGetcwd\(\)--Get Current Directory](#)

Example

The following example determines the current directory:

```
#include <unistd.h>
#include <stdio.h>

main()
{
    char cwd[1024];

    if (chdir("/tmp") != 0)
        perror("chdir() error");
    else
    {
        if (getcwd(cwd, sizeof(cwd)) == NULL)
            perror("getcwd() error");
        else
            printf("current working directory is: %s\n", cwd);
    }
}
```

Output:

```
current working directory is: /tmp
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

getegid()--Get Effective Group ID

Syntax

```
#include <unistd.h>

gid_t getegid(void);

Threadsafe: Yes
```

The **getegid()** function returns the effective group ID (*gid*) of the calling thread. The effective *gid* is the group ID under which the thread is currently running. The effective *gid* of a thread may change while the thread is running.

Parameters

None.

Authorities

No authorization is required.

Return Value

> 0

getegid() was successful. The value returned represents the effective *gid*.

>= 0

getegid() was successful. If there is no *gid*, the user ID has no group profile associated with it and returns 0. Otherwise, if there is a group profile, the API returns the *gid* of the group profile.

-1

getegid() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **getegid()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EAGAIN]

Internal object compressed. Try again.

[EDAMAGE]

The user profile associated with the thread *gid* or an internal system object is damaged.

[*ENOMEM*]

The user profile associated with the thread *gid* has exceeded its storage limit.

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))

Example

The following example gets the effective gid.

```
#include <unistd.h>

main()
{
    gid_t ef_gid;

    if (-1 == (ef_gid = getegid(void)))
        perror("getegid() error.");
    else
        printf("The effective gid is: %u\n", ef_gid);
}
```

Output:

```
The effective gid is: 75
```

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

geteuid()--Get Effective User ID

Syntax

```
#include <unistd.h>

uid_t geteuid(void);

Threadsafe: Yes
```

The **geteuid()** function returns the effective user ID (uid) of the calling thread. The effective uid is the user ID under which the thread is currently running. The effective uid of a thread may change while the thread is running.

Parameters

None.

Authorities

No authorization is required.

Return Value

0 or > 0

geteuid() was successful. The value returned represents the effective uid.

-1

geteuid() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **geteuid()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EAGAIN]

Internal object compressed. Try again.

[EDAMAGE]

The user profile associated with the thread uid or an internal system object is damaged.

[ENOMEM]

The user profile associated with the thread uid has exceeded its storage limit.

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))

Example

The following example gets the effective uid.

```
#include <unistd.h>

main()
{
    uid_t ef_uid;

    if (-1 == (ef_uid = geteuid(void)))
        perror("geteuid() error.");
    else
        printf("The effective uid is: %u\n", ef_uid);
}
```

Output:

```
The effective uid is: 1957
```

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

getgid()--Get Real Group ID

Syntax

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

```
gid_t getgid(void);
```

Threadsafe: Yes

The **getgid()** function returns the real group ID (*gid*) of the calling thread. The real *gid* is the group ID under which the thread was created.

Note: When a user profile swap is done with the QWTSETP API prior to running the **getgid()** function, the *gid* for the current profile is returned.

Parameters

None.

Authorities

No authorization is required.

Return Value

> 0

getgid() was successful. The value returned represents the *gid*.

≥ 0

getgid() was successful. If there is no *gid*, the user ID has no group profile associated with it and returns 0. Otherwise, if there is a group profile, the API returns the *gid* of the group profile.

-1

getgid() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **getgid()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EAGAIN]

Internal object compressed. Try again.

[EDAMAGE]

The user profile associated with the thread *gid* or an internal system object is damaged.

[*ENOMEM*]

The user profile associated with the thread *gid* has exceeded its storage limit.

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))

Example

The following example gets the real gid.

```
#include <unistd.h>

main()
{
    gid_t gid;

    if (-1 == (gid = getgid(void)))
        perror("getgid() error.");
    else
        printf("The real gid is: %u\n", gid);
}
```

Output:

```
The real gid is: 75
```

getgrgid()--Get Group Information Using Group ID

Syntax

```
#include <grp.h>

struct group *getgrgid(gid_t gid);

Threadsafe: No
```

The **getgrgid()** function returns a pointer to an object of type struct group containing an entry from the user database with a matching *gid*.

Parameters

gid

(Input) Group ID.

Authorities

*READ authority is required to the user profile associated with the *gid*. If the user does not have *READ authority, only the name of the group and the group ID values are returned.

Return Value

*struct group **

getgrgid() was successful. The return value points to static data of the format struct group, which is defined in the **grp.h** header file. This storage is overwritten on each call to this function. This static storage area is also used by the **getgrnam()** function. The struct group has the following elements:

char *	gr_name	Name of the group
gid_t	gr_gid	Group ID
char **	gr_mem	A null-terminated list of pointers to the individual member profile names. If the group profile does not have any members or if the caller does not have *READ authority to the group profile, the list will be empty.

NULL pointer

getgrgid() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If `getgrgid()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EAGAIN]

The user profile associated with the *gid* is currently locked by another process.

[EC2]

Detected pointer that is not valid.

[EINVAL]

Value is not valid. Check the job log for messages.

[ENOENT]

The user profile associated with the *gid* was not found.

[ENOMEM]

The user profile associated with the *gid* has exceeded its storage limit.

[ENOSPC]

Machine storage limit exceeded.

Related Information

- The `<grp.h>` file (see [Header Files for UNIX-Type Functions](#))
- [getgrgid_r\(\)--Get Group Information Using Group ID](#)

Example

The following example gets the group information for the *gid* of 91. The group name is GROUP1. There are two group members, CLIFF and PATRICK.

```
#include <grp.h>
#include <stdio.h>

main()
{
    struct group *grp;
    short int    lp;

    if (NULL == (grp = getgrgid(91)))
        perror("getgrgid() error.");
    else
    {
        printf("The group name is: %s\n", grp->gr_name);
        printf("The gid      is: %u\n", grp->gr_gid);
        for (lp = 1; NULL != *(grp->gr_mem); lp++, (grp->gr_mem)++)
            printf("Group member %d is: %s\n", lp, *(grp->gr_mem));
    }
}
```

```
}
```

Output:

```
The group name is: GROUP1  
The gid      is: 91  
Group member 1 is: CLIFF  
Group member 2 is: PATRICK
```

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

getgrgid_r()--Get Group Information Using Group ID

Syntax

```
#include <sys/types.h>
#include <grp.h>

int getgrgid_r(gid_t gid, struct group *grp,
char *buffer, size_t bufsize, struct group
**result);
```

Threadsafe: Yes

The **getgrgid_r()** function updates the group structure pointed to by *grp* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *gid*.

Parameters

gid

(Input) Group ID.

grp

(Input) A pointer to a group structure.

buffer

(Input) A pointer to a buffer from which memory is allocated to hold storage areas referenced by the group structure *grp*.

bufsize

(Input) The size of buffer in bytes.

result

(Input) A pointer to a location in which a pointer to the updated group structure is stored. If an error occurs or if the requested entry cannot be found, a NULL pointer is stored in this location.

The struct group, which is defined in the **grp.h** header file, has the following elements:

char *	gr_name	Name of the group
gid_t	gr_gid	Group ID
char **	gr_mem	A null-terminated list of pointers to the individual member profile names. If the group profile does not have any members or if the caller does not have *READ authority to the group profile, the list will be empty.

Authorities

*READ authority is required to the user profile associated with the *gid*. If the user does not have *READ authority, only the name of the group and the group ID values are returned.

Return Value

0

getgrgid_r was successful.

Any other value

Failure: The return value contains an error number indicating the error.

Error Conditions

If **getgrgid_r()** is not successful, the return value usually indicates one of the following errors. Under some conditions, the value could indicate an error other than those listed here.

[EAGAIN]

The user profile associated with the *gid* is currently locked by another process.

[EC2]

Detected pointer that is not valid.

[EINVAL]

Value is not valid. Check the job log for messages.

[ENOENT]

The user profile associated with the *gid* was not found.

[ENOMEM]

The user profile associated with the *gid* has exceeded its storage limit.

[ENOSPC]

Machine storage limit exceeded.

[ERANGE]

Insufficient storage was supplied by *buffer* and *bufsize* to contain the data to be referenced by the resulting group structure.

Related Information

- The <grp.h> file [Header Files for UNIX-Type Functions](#)(see)
- [getgrgid\(\)--Get Group Information Using Group ID](#)

Example

The following example gets the group information for the gid of 91. The group name is GROUP1. There are two group members, CLIFF and PATRICK.

```
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>
#include <errno.h>

main()
{ short int lp;
  struct group grp;
  struct group * grpPtr=&grp;
  struct group * tempGrpPtr;
  char grpbuffer[200];
  int  grpstrlen = sizeof(grpbuffer);

  if ((getgrgid_r(91,grpPtr,grpbuffer,grpstrlen,&tempGrpPtr))!=0)
    perror("getgrgid_r() error.");
  else
  {
    printf("\nThe group name is: %s\n", grp.gr_name);
    printf("The gid          is: %u\n", grp.gr_gid);
    for (lp = 1; NULL != *(grp.gr_mem); lp++, (grp.gr_mem)++)
      printf("Group Member %d is: %s\n", lp, *(grp.gr_mem));
  }
}
```

Output:

```
The group name is: GROUP1
The gid          is: 91
Group member 1 is: CLIFF
Group member 2 is: PATRICK
```

getgrgid_r_ts64()--Get Group Information Using Group ID

Syntax

```
#include <sys/types.h>
#include <grp.h>

int getgrgid_r_ts64(
    gid_t gid,
    struct group * __ptr64 grp,
    char * __prt64 buffer,
    size_t bufsize,
    struct group * __ptr64 * __ptr64 result);
```

Service Program Name: QSYPAPI64

Default Public Authority: *USE

Threadsafe: Yes

The **getgrgid_r_ts64()** function updates the group structure pointed to by *grp* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *gid*. **getgrgid_r_ts64()** differs from **getgrgid_r()** in that it accepts 8-byte teraspace pointers.

For a discussion of the parameters, authorities required, return values, related information, usage notes, and an example for the **getgrgid_r()** API, see [getgrgid_r\(\)--Get Group Information Using Group ID](#).

API introduced: V5R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

getgrnam()--Get Group Information Using Group Name

Syntax

```
#include <grp.h>

struct group *getgrnam(const char *name);

Threadsafe: No
```

The **getgrnam()** function returns a pointer to an object of type `struct group` containing an entry from the user database with a matching *name*.

Parameters

name

(Input) A pointer to a group profile name.

Authorities

*READ authority is required to the user profile associated with the *name*. If the user does not have *READ authority, only the name of the group and the group ID values are returned.

Return Value

*struct group **

getgrnam() was successful. The return value points to static data of the format `struct group`, which is defined in the **grp.h** header file. This storage is overwritten on each call to this function. This static storage area is also used by the **getgrgid()** function. The `struct group` has the following elements:

<code>char *</code>	<code>gr_name</code>	Name of the group
<code>gid_t</code>	<code>gr_gid</code>	Group ID
<code>char **</code>	<code>gr_mem</code>	A null-terminated list of pointers to the individual member profile names. If the group profile does not have any members or if the caller does not have *READ authority to the group profile, the list will be empty.

NULL pointer

getgrnam was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If `getgrnam()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EAGAIN]

The user profile associated with the *name* is currently locked by another process.

[EC2]

Detected pointer that is not valid.

[EDAMAGE]

The user profile associated with the group name or an internal system object is damaged.

[EINVAL]

Value is not valid. Check the job log for messages.

[ENOENT]

The user profile associated with the *name* was not found or the profile name specified is not a group profile.

[EUNKNOWN]

Unknown system state. Check the job log for a CPF9872 message.

Related Information

- The `<grp.h>` file (see [Header Files for UNIX-Type Functions](#))
- [getgrnam_r\(\)--Get Group Information Using Group Name](#)

Example

The following example gets the group information for the group GROUP1. The gid is 91. There are two group members, CLIFF and PATRICK.

```
#include <grp.h>
#include <stdio.h>

main()
{
    struct group *grp;
    short int    lp;

    if (NULL == (grp = getgrnam("GROUP1")))
        perror("getgrnam() error.");
    else
    {
        printf("The group name is: %s\n", grp->gr_name);
        printf("The gid      is: %u\n", grp->gr_gid);
        for (lp = 1; NULL != *(grp->gr_mem); lp++, (grp->gr_mem)++)
            printf("Group member %d is: %s\n", lp, *(grp->gr_mem));
    }
}
```

```
}
```

Output:

```
The group name is: GROUP1  
The gid      is: 91  
Group member 1 is: CLIFF  
Group member 2 is: PATRICK
```

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

getgrnam_r()--Get Group Information Using Group Name

Syntax

```
#include <sys/types.h>
#include <grp.h>

int getgrnam_r(const char *name, struct group *grp,
char *buffer, size_t bufsize, struct group
**result);
```

Threadsafe: Yes

The **getgrnam_r()** function updates the group structure pointed to by *grp* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with matching *name*.

Parameters

name

(Input) A pointer to a group profile name.

grp

(Input) A pointer to a group structure.

buffer

(Input) A pointer to a buffer from which memory is allocated to hold storage areas referenced by the group structure *grp*.

bufsize

(Input) The size of *buffer* in bytes.

result

(Input) A pointer to a location in which a pointer to the updated group structure is stored. If an error occurs or the requested entry cannot be found, a NULL pointer is stored in this location.

The struct group, which is defined in the **grp.h** header file, has the following elements:

char *	gr_name	Name of the group
gid_t	gr_gid	Group ID
char **	gr_mem	A null-terminated list of pointers to the individual member profile names. If the group profile does not have any members or if the caller does not have *READ authority to the group profile, the list will be empty.

Authorities

*READ authority is required to the user profile associated with the *name*. If the user does not have *READ authority, only the name of the group and the group ID values are returned.

Return Value

0

getgrnam_r was successful.

Any other value

Failure: The return value contains an error number indicating the error.

Error Conditions

If **getgrnam_r()** is not successful, the return value usually indicates one of the following errors. Under some conditions, the value could indicate an error other than those listed here.

[EAGAIN]

The user profile associated with the *name* is currently locked by another process.

[EC2]

Detected pointer that is not valid.

[EDAMAGE]

The user profile associated with the group name or an internal system object is damaged.

[EINVAL]

Value is not valid. Check the job log for messages.

[ENOENT]

The user profile associated with the *name* was not found or the profile name specified is not a group profile.

[ERANGE]

Insufficient storage was supplied by *buffer* and *bufsize* to contain the data to be referenced by the resulting group structure.

[EUNKNOWN]

Unknown system state. Check the job log for a CPF9872 message.

Related Information

- The <grp.h> file (see)
- [getgrnam\(\)--Get Group Information Using Group Name](#)

Example

The following example gets the group information for the group GROUP1. The gid is 91. There are two group members, CLIFF and PATRICK.

```
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>
#include <errno.h>

main()
{ short int lp;
  struct group grp;
  struct group * grpPtr=&grp;
  struct group * tempGrpPtr;
  char grpbuffer[200];
  int  grpstrlen = sizeof(grpbuffer);

  if ((getgrnam_r("GROUP1",grpPtr,grpbuffer,grpstrlen,&tempGrpPtr))!=0)
    perror("getgrnam_r() error.");
  else
  {
    printf("\nThe group name is: %s\n", grp.gr_name);
    printf("The gid          is: %u\n", grp.gr_gid);
    for (lp = 1; NULL != *(grp.gr_mem); lp++, (grp.gr_mem)++)
      printf("Group Member %d is: %s\n", lp, *(grp.gr_mem));
  }
}
```

Output:

```
The group name is: GROUP1
The gid          is: 91
Group member 1 is: CLIFF
Group member 2 is: PATRICK
```

getgrnam_r_ts64()--Get Group Information Using Group Name

Syntax

```
#include <sys/types.h>
#include <grp.h>

int getgrnam_r_ts64(
    const char * __ptr64 name,
    struct group * __ptr64 grp,
    char * __ptr64 buffer,
    size_t bufsize,
    struct group * __ptr64 * __ptr64 result);
```

Service Program Name: QSYPAPI64

Default Public Authority: *USE

Threadsafe: Yes

The **getgrnam_r_ts64()** function updates the group structure pointed to by *grp* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *name*. **getgrnam_r_ts64()** differs from **getgrnam_r()** in that it accepts 8-byte teraspace pointers.

For a discussion of the parameters, authorities required, return values, related information, usage notes, and an example for the **getgrnam_r()** API, see [getgrnam_r\(\)--Get Group Information Using Group Name](#).

API introduced: V5R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

getgroups()--Get Group IDs

Syntax

```
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[])
```

Threadsafe: No

If the *gidsetsize* argument is zero, **getgroups()** returns the number of group IDs associated with the calling thread without modifying the array pointed to by the *grouplist* argument. The number of group IDs includes the effective group ID and the supplementary group IDs. Otherwise, **getgroups()** fills in the array *grouplist* with the effective group ID and supplementary group IDs of the calling thread and returns the actual number of group IDs stored. The values of array entries with indexes larger than or equal to the returned value are undefined.

Parameters

gidsetsize

(Input) The number of elements in the supplied array *grouplist*.

grouplist

(Output) The effective group ID and supplementary group IDs. The first element in *grouplist* is the effective group ID.

Authorities

No authorization is required.

Return Value

- 0 or > 0 **getgroups()** was successful. If the *gidsetsize* argument is 0, the number of group IDs is returned. This number includes the effective group ID and supplementary group IDs. If *gidsetsize* is greater than 0, the array *grouplist* is filled with the effective group ID and supplementary group IDs of the calling thread and the return value represents the actual number of group IDs stored.
- -1 **getgroups()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If `getgroups()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EINVAL]

The *gidsetsize* argument is not equal to zero and is less than the number of [»](#)group IDs.[«](#)

Usage Notes

This function can be used in two different ways. First, if called with *gidsetsize* equal to 0, it is used to return the number of groups associated with a thread. Second, if called with *gidsetsize* not equal to 0, it is used to return a list of the `gids` representing the [»](#)effective [«](#)and supplementary groups associated with a thread. In this case, the *gidsetsize* argument represents how much space is available in the *grouplist* argument.

The calling routine can choose to call this function with *gidsetsize* equal to 0 to determine how much space to allocate for a second call to this function. The second call returns the values. The following is an example of this method:

```
int numgroups;
gid_t *grouplist;

numgroups = getgroups(0, NULL);
grouplist = (gid_t *) calloc( numgroups, sizeof(gid_t) );
if (getgroups( numgroups, grouplist) != -1) {
    .
    .
}
```

Alternatively, the calling routine can choose to create enough space for `NGROUPS_MAX` entries to ensure enough space is available for the maximum possible number of entries that may be returned. This introduces the possibility of wasted space. The following is an example of this method:

```
int numgroups;
gid_t grouplist[ NGROUPS_MAX ];

if ( getgroups( NGROUPS_MAX, grouplist ) != -1 ) {
    .
    .
}
```

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))

getpwnam()--Get User Information for User Name

Syntax

```
#include <pwd.h>

struct passwd *getpwnam(const char *name);

Threadsafe: No
```

The **getpwnam()** function returns a pointer to an object of type struct passwd containing an entry from the user database with a matching *name*.

Parameters

name

(Input) A pointer to a user profile name.

Authorities

*READ authority is required to the user profile associated with the *name*. If the user does not have *READ authority, only the user name, user ID, and group ID values are returned.

Note: Adopted authority is not used.

Return Value

*struct passwd **

getpwnam() was successful. The return value points to static data of the format struct passwd, which is defined in the **pwd.h** header file. This storage is overwritten on each call to this function. This static storage area is also used by the **getpwuid()** function. The struct passwd has the following elements:

char *	pw_name	User name
uid_t	pw_uid	User ID
uid_t	pw_gid	Group ID of the user's first group. If the user does not have a first group, the gid value will be set to 0.
char *	pw_dir	Initial working directory. If the user does not have *READ authority to the user profile, the pw_dir pointer will be set to NULL.
char *	pw_shell	Initial user program. If the user does not have *READ authority to the user profile, the pw_shell pointer will be set to NULL.

NULL pointer

getpwnam() was not successful. The *errno* global variable is set to indicate the error.

See [QlgGetpwnam\(\)--Get User Information for User Name \(using NLS-enabled path name\)](#) for a description and an example where the path name is returned in any CCSID.

Error Conditions

If **getpwnam()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EAGAIN]

The user profile associated with the *name* is currently locked by another process.

[EC2]

Detected pointer that is not valid.

[EINVAL]

Value is not valid. Check the job log for messages.

[ENOENT]

The user profile associated with the *name* was not found.

[ENOMEM]

The user profile associated with the *uid* has exceeded its storage limit or is unable to allocate memory.

[EUNKNOWN]

Unknown system state. Check the job log for a CPF9872 message. If there is no message, verify that the home directory field in the user profile can be displayed.

Usage Notes

The initial working directory is returned in the CCSID value of the job.

Related Information

- The <pwd.h> file (see [Header Files for UNIX-Type APIs](#))
- [getpwnam_r\(\)--Get User Information for User Name](#)
- [QlgGetpwnam\(\)--Get User Information for User Name \(using NLS-enabled path name\)](#)

Example

The following example gets the user database information for the user name of MYUSER. The uid is 22. The gid of MYUSER's first group is 1012. The initial directory is /home/MYUSER. The initial user program is *LIBL/QCMD.

```
#include <pwd.h>

main()
{
    struct passwd *pd;

    if (NULL == (pd = getpwnam("MYUSER")))
        perror("getpwnam() error.");
    else
    {
        printf("The user name is: %s\n", pd->pw_name);
        printf("The user id   is: %u\n", pd->pw_uid);
        printf("The group id  is: %u\n", pd->pw_gid);
        printf("The initial directory is:   %s\n", pd->pw_dir);
        printf("The initial user program is: %s\n", pd->pw_shell);
    }
}
```

Output:

```
The user name is: MYUSER
The user id   is: 22
The group id  is: 1012
The initial directory is:   /home/MYUSER
The initial user program is: *LIBL/QCMD
```

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

getpwnam_r()--Get User Information for User Name

Syntax

```
#include <sys/types.h>
#include <pwd.h>

int getpwnam_r(const char *name, struct passwd
*pwd, char *buffer, size_t bufsize,
struct passwd **result);
```

Service Program Name: QSYPAPI

Default Public Authority: *USE

Threadsafe: Yes

The **getpwnam_r()** function updates the *passwd* structure pointed to by *pwd* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *name*.

Parameters

name

(Input) A pointer to a user profile name.

pwd

(Input) A pointer to a *passwd* structure.

buffer

(Input) A pointer to a buffer from which memory is allocated to hold storage areas referenced by the structure *pwd*.

bufsize

(Input) The size of *buffer* in bytes.

result

(Input) A pointer to a location in which a pointer to the updated *passwd* structure is stored. If an error occurs or if the requested entry cannot be found, a NULL pointer is stored in this location.

The struct *passwd*, which is defined in the **pwd.h** header file, has the following elements:

char *	pw_name	User name
uid_t	pw_uid	User ID

uid_t	pw_gid	Group ID of the user's first group. If the user does not have a first group, the GID value will be set to 0.
char *	pw_dir	Initial working directory. If the user does not have *READ authority to the user profile, the pw_dir pointer will be set to NULL.
char *	pw_shell	Initial user program. If the user does not have *READ authority to the user profile, the pw_shell will be set to NULL.

See [QlgGetpwnam_r\(\)--Get User Information for User Name \(using NLS-enabled path name\)](#) for a description and an example where the path name is returned in any CCSID. Go to `_r` version

Authorities

*READ authority is required to the user profile associated with the *name*. If the user does not have *READ authority, only the user name, user ID, and group ID values are returned.

Note: Adopted authority is not used.

Return Value

0

`getpwnam_r` was successful.

Any other value

Failure: The return value contains an error number indicating the error.

Error Conditions

If `getpwnam_r()` is not successful, the return value usually indicates one of the following errors. Under some conditions, the value could indicate an error other than those listed here.

<code>[EAGAIN]</code>	The user profile associated with the <i>name</i> is currently locked by another process.
<code>[EC2]</code>	Detected pointer that is not valid.
<code>[EINVAL]</code>	Value is not valid. Check the job log for messages.
<code>[ENOENT]</code>	The user profile associated with the <i>name</i> was not found.
<code>[ENOMEM]</code>	The user profile associated with the <i>uid</i> has exceeded its storage limit or is unable to allocate memory.
<code>[ERANGE]</code>	Insufficient storage was supplied through <i>buffer</i> and <i>bufsize</i> to contain the data to be referenced by the resulting group structure.
<code>[EUNKNOWN]</code>	Unknown system state. Check the job log for a CPF9872 message. If there is no message, verify that the home directory field in the user profile can be displayed.

Usage Notes

The initial working directory is returned in the CCSID value of the job.

Related Information

- The `<pwd.h>` file (see [Header Files for UNIX-Type Functions](#))
- [getpwnam\(\)](#)--Get User Information for User Name

Example

The following example gets the user database information for the user name of MYUSER. The UID is 22. The GID of MYUSER's first group is 1012. The initial directory is /home/MYUSER. The initial user program is *LIBL/QCMD.

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <errno.h>

main()
{
    struct passwd pd;
    struct passwd* pwdptr=&pd;
    struct passwd* tempPwdPtr;
    char pwdbuffer[200];
    int pwdbuflen = sizeof(pwdbuffer);

    if ((getpwnam_r("MYUSER",pwdptr,pwdbuffer,pwdbuflen,&tempPwdPtr))!=0)
        perror("getpwnam_r() error.");
    else
    {
        printf("\nThe user name is: %s\n", pd.pw_name);
        printf("The user id   is: %u\n", pd.pw_uid);
        printf("The group id  is: %u\n", pd.pw_gid);
        printf("The initial directory is:   %s\n", pd.pw_dir);
        printf("The initial user program is: %s\n", pd.pw_shell);
    }
}
```

Output:

```
The user name is: MYUSER
The user ID   is: 22
The group ID  is: 1012
The initial directory is:   /home/MYUSER
```

The initial user program is: *LIBL/QCMD

API introduced: V4R4

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

getpwnam_r_ts64()--Get User Information for User Name

Syntax

```
#include <sys/types.h>
#include <pwd.h>

int getpwnam_r_ts64(
    const char * __ptr64 name,
    struct passwd * __ptr64 pwd,
    char * __ptr64 buffer,
    size_t bufsize,
    struct passwd * __ptr64 * __ptr64 result);
```

Service Program Name: QSYPAPI64

Default Public Authority: *USE

Threadsafe: Yes

The **getpwnam_r_ts64()** function updates the *passwd* structure pointed to by *pwd* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *name*. **getpwnam_r_ts64()** differs from **getpwnam_r()** in that it accepts 8-byte teraspace pointers.

For a discussion of the parameters, authorities required, return values, related information, usage notes, and an example for the **getpwnam_r()** API, see [getpwnam_r\(\)](#)--Get User Information for User Name.

API introduced: V5R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

getpwuid()--Get User Information for User ID

Syntax

```
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);
```

Threadsafe: No

The **getpwuid()** function returns a pointer to an object of type struct passwd containing an entry from the user database with a matching *uid*.

Parameters

uid

(Input) User ID.

Authorities

*READ authority is required to the user profile associated with the *uid*. If the user does not have *READ authority, only the user name, user ID, and group ID values are returned.

Note: Adopted authority is not used.

Return Value

*struct passwd **

getpwuid() was successful. The return value points to static data of the format struct passwd, which is defined in the **pwd.h** header file. This storage is overwritten on each call to this function. This static storage area is also used by the **getpwnam()** function. The struct passwd has the following elements:

char *	pw_name	User name
uid_t	pw_uid	User ID
uid_t	pw_gid	Group ID of the user's first group. If the user does not have a first group, the gid value will be set to 0.
char *	pw_dir	Initial working directory. If the user does not have *READ authority to the user profile, the pw_dir pointer will be set to NULL.
char *	pw_shell	Initial user program. If the user does not have *READ authority to the user profile, the pw_shell pointer will be set to NULL.

NULL pointer

`getpwuid()` was not successful. The *errno* global variable is set to indicate the error.

See [QlgGetpwuid\(\)--Get User Information for User ID \(using NLS-enabled path name\)](#) for a description and an example where the path name is returned in any CCSID.

Error Conditions

If `getpwuid()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EAGAIN]

The user profile associated with the *uid* is currently locked by another process.

[EC2]

Detected pointer that is not valid.

[EINVAL]

Value is not valid. Check the job log for messages.

[ENOENT]

The user profile associated with *uid* was not found.

[ENOMEM]

The user profile associated with the *uid* has exceeded its storage limit or is unable to allocate memory.

[ENOSPC]

Machine storage limit exceeded.

[EUNKNOWN]

Unknown system state. Check the job log for a CPF9872 message. If there is no message, verify that the home directory field in the user profile can be displayed.

Usage Notes

The initial working directory is returned in the CCSID value of the job.

Related Information

- The `<pwd.h>` file (see [Header Files for UNIX-Type Functions](#))
- [getpwuid_r\(\)--Get User Information for User ID](#)
- [QlgGetpwuid\(\)--Get User Information for User ID \(using NLS-enabled path name\)](#)

Example

The following example gets the user database information for the uid of 22. The user name is MYUSER. The gid of MYUSER's first group is 1012. The initial directory is /home/MYUSER. The initial user program is *LIBL/QCMD.

```
#include <pwd.h>

main()
{
    struct passwd *pd;

    if (NULL == (pd = getpwuid(22)))
        perror("getpwuid() error.");
    else
    {
        printf("The user name is: %s\n", pd->pw_name);
        printf("The user id   is: %u\n", pd->pw_uid);
        printf("The group id  is: %u\n", pd->pw_gid);
        printf("The initial directory is:   %s\n", pd->pw_dir);
        printf("The initial user program is: %s\n", pd->pw_shell);
    }
}
```

Output:

```
The user name is: MYUSER
The user id   is: 22
The group id  is: 1012
The initial directory is:   /home/MYUSER
The initial user program is: *LIBL/QCMD
```

getpwuid_r()--Get User Information for User ID

Syntax

```
#include <sys/types.h>
#include <pwd.h>

int getpwuid_r(uid_t uid, struct passwd *pwd,
char *buffer, size_t bufsize, struct passwd
**result);
```

Service Program Name: QSYPAPI

Default Public Authority: *USE

Threadsafe: Yes

The **getpwuid_r()** function updates the *passwd* structure pointed to by *pwd* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *uid*.

Parameters

uid

(Input) User ID.

pwd

(Input) A pointer to a struct passwd.

buffer

(Input) A pointer to a buffer from which memory is allocated to hold storage areas referenced by the structure passwd.

bufsize

(Input) The size of *buffer* in bytes.

result

(Input) A pointer to a location in which a pointer to the updated passwd structure is stored. If an error occurs or if the requested entry cannot be found, a NULL pointer is stored in this location.

The struct passwd, which is defined in the **pwd.h** header file, has the following elements:

char *	pw_name	User name
uid_t	pw_uid	User ID

uid_t	pw_gid	Group ID of the user's first group. If the user does not have a first group, the GID value will be set to 0.
char *	pw_dir	Initial working directory. If the user does not have *READ authority to the user profile, the pw_dir pointer will be set to NULL.
char *	pw_shell	Initial user program. If the user does not have *READ authority to the user profile, the pw_shell pointer will be set to NULL.

See [QlgGetpwuid_r\(\)--Get User Information for User ID \(using NLS-enabled path name\)](#) for a description and an example where the path name is returned in any CCSID.

Authorities

*READ authority is required to the user profile associated with the *uid*. If the user does not have *READ authority, only the user name, user ID, and group ID values are returned.

Note: Adopted authority is not used.

Return Value

0

getpwuid_r() was successful.

Any other value

Failure: The return value contains an error number indicating the error.

Error Conditions

If **getpwuid_r()** is not successful, the error value usually indicates one of the following errors. Under some conditions, the value could indicate an error other than those listed here.

<i>[EAGAIN]</i>	The user profile associated with the <i>uid</i> is currently locked by another process.
<i>[EC2]</i>	Detected pointer that is not valid.
<i>[EINVAL]</i>	Value is not valid. Check the job log for messages.
<i>[ENOENT]</i>	The user profile associated with the <i>uid</i> was not found.
<i>[ENOMEM]</i>	The user profile associated with the <i>uid</i> has exceeded its storage limit or is unable to allocate memory.
<i>[ENOSPC]</i>	Machine storage limit exceeded.
<i>[ERANGE]</i>	Insufficient storage was supplied through <i>buffer</i> and <i>bufsize</i> to contain the data to be referenced by the resulting group structure.

[EUNKNOWN]

Unknown system state. Check the job log for a CPF9872 message. If there is no message, verify that the home directory field in the user profile can be displayed.

Usage Notes

The initial working directory is returned in the CCSID value of the job.

Related Information

- The <pwd.h> file (see [Header Files for UNIX-Type Functions](#))
- [getpwuid\(\)](#)--Get User Information for User ID

Example

The following example gets the user database information for the UID of 22. The user name is MYUSER. The GID of MYUSER's first group is 1012. The initial directory is /home/MYUSER. The initial user program is *LIBL/QCMD.

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <errno.h>

main()
{
    struct passwd pd;
    struct passwd* pwdptr=&pd;
    struct passwd* tempPwdPtr;
    char pwdbuffer[200];
    int pwdbuflen = sizeof(pwdbuffer);

    if ((getpwuid_r(22,pwdptr,pwdbuffer,pwdbuflen,&tempPwdPtr))!=0)
        perror("getpwuid_r() error.");
    else
    {
        printf("\nThe user name is: %s\n", pd.pw_name);
        printf("The user id   is: %u\n", pd.pw_uid);
        printf("The group id  is: %u\n", pd.pw_gid);
        printf("The initial directory is:   %s\n", pd.pw_dir);
        printf("The initial user program is: %s\n", pd.pw_shell);
    }
}
```

Output:

The user name is: MYUSER
The user ID is: 22
The group ID is: 1012
The initial directory is: /home/MYUSER
The initial user program is: *LIBL/QCMD

API introduced: V4R4

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

getpwuid_r_ts64()--Get User Information for User ID

Syntax

```
#include <sys/types.h>
#include <pwd.h>

int getpwuid_r_ts64(
    uid_t uid,
    struct passwd * __ptr64 pwd,
    char * __ptr64 buffer,
    size_t bufsize,
    struct passwd * __ptr64 * __ptr64 result);
```

Service Program Name: QSYPAPI64

Default Public Authority: *USE

Threadsafe: Yes

The **getpwuid_r_ts64()** function updates the *passwd* structure pointed to by *pwd* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *uid*. **getpwuid_r_ts64()** differs from **getpwuid_r()** in that it accepts 8-byte teraspace pointers.

For a discussion of the parameters, authorities required, return values, related information, usage notes, and an example for the **getpwuid_r()** API, see [getpwuid_r\(\)--Get User Information for User ID](#).

API introduced: V5R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

getuid()--Get Real User ID

Syntax

```
#include <unistd.h>

uid_t getuid(void);

Threadsafe: Yes
```

The **getuid()** function returns the real user ID (uid) of the calling thread. The real uid is the user ID under which the thread was created.

Note: When a user profile swap is done with the QWTSETP API prior to running the `getuid()` function, the uid for the current profile is returned.

Parameters

None.

Authorities

No authorization is required.

Return Value

0 or > 0

getuid() was successful. The value returned represents the *uid*.

-1

getuid() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **getuid()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EAGAIN]

Internal object compressed. Try again.

[EDAMAGE]

The user profile associated with the thread *uid* or an internal system object is damaged.

[ENOMEM]

The user profile associated with the thread uid has exceeded its storage limit.

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))

Example

The following example gets the real uid.

```
#include <unistd.h>

main()
{
    uid_t uid;

    if (-1 == (uid = getuid(void)))
        perror("getuid() error.");
    else
        printf("The real uid is: %u\n", uid);
}
```

Output:

```
The real uid is: 1957
```

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

ioctl()--Perform I/O Control Request

Syntax

```
#include <sys/types.h>
#include <sys/ioctl.h>

int ioctl(int descriptor,
          unsigned long request,
          ...);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see [Usage Notes](#).

The **ioctl()** function performs control functions (requests) on a descriptor.

Parameters

descriptor

(Input) The descriptor on which the control request is to be performed.

request

(Input) The request that is to be performed on the *descriptor*.

...

(Input) A variable number of optional parameters that are dependent on the request.

The *ioctl()* requests that are supported are:

FIOASYNC

Set or clear the flag that allows the receipt of asynchronous I/O signals (SIGIO).

The third parameter represents a pointer to an integer flag. A nonzero value sets the socket to generate SIGIO signals, while a zero value sets the socket to not generate SIGIO signals. Note that before the SIGIO signals can be delivered, you must use either the FIOSETOWN or SIOCSPGRP *ioctl()* request, or the F_SETOWN *fcntl()* command to set a process ID or a process group ID to indicate what process or group of processes will receive the signal. Once conditioned to send SIGIO signals, a socket will generate SIGIO signals whenever certain significant conditions change on the socket. For example, SIGIO will be generated when normal data arrives on the socket, when out-of-band data arrives on the socket (in addition to the SIGURG signal), when an error occurs on the socket, or when end-of-file is received on the socket. It is also generated when a connection request is received on the socket (if it is a socket on which the *listen()* verb has been done). Also note that a socket can be set to generate the SIGIO signal by using the *fcntl()* command F_SETFL with a flag value specifying FASYNC.

FIOCCSID

Return the coded character set ID (CCSID) associated with the open instance represented by the descriptor and the CCSID associated with the object. The third parameter represents a pointer to the structure Qp01FIOCCSID, which is defined in <sys/ioctl.h>. This information may be necessary to correctly manipulate data read from or written to a file opened in another process.

If the open instance represented by the descriptor is in binary mode (the *open()* did not specify the O_TEXTDATA open flag), the open instance CCSID returned is equal to the object CCSID returned.

- FIOGETOWN* Get the process ID or process group ID that is to receive the SIGIO and SIGURG signals.
- The third parameter represents a pointer to a signed integer that will contain the process ID or the process group ID to which the socket is currently sending asynchronous signals such as SIGURG. A process ID is returned as a positive integer, and a process group ID is specified as a negative integer. A 0 value returned indicates that no asynchronous signals can be generated by the socket. A positive or a negative value indicates that the socket has been set to generate SIGURG signals.
- FIONBIO* Set or clear the nonblocking I/O flag (O_NONBLOCK oflag). The third parameter represents a pointer to an integer flag. A nonzero value sets the nonblocking I/O flag for the descriptor; a zero value clears the flag.
- FIONREAD* Return the number of bytes available to be read. The third parameter represents a pointer to an integer that is set to the number of bytes available to be read.
- FIOSETOWN* Set the process ID or process group ID that is to receive the SIGIO and SIGURG signals.
- The third parameter represents a pointer to a signed integer that contains the process ID or the process group ID to which the socket should send asynchronous signals such as SIGURG. A process ID is specified as a positive integer, and a process group ID is specified as a negative integer. Specifying a 0 value resets the socket such that no asynchronous signals are delivered. Specifying a process ID or a process group ID requests that sockets begin sending the SIGURG signal to the specified ID when out-of-band data arrives on the socket.
- SIOCADDRT* Add an entry to the interface routing table. Valid for sockets with address family of AF_INET.

The third parameter represents a pointer to the structure **rtentry**, which is defined in `<net/route.h>`:

```

struct rtentry [
    struct sockaddr rt_dst;
    struct sockaddr rt_mask;
    struct sockaddr rt_gateway;
    int rt_mtu;
    u_short rt_flags;
    u_short rt_refcnt;
    u_char rt_protocol;
    u_char rt_TOS;
    char rt_if[IFNAMSIZ];
];

```

The *rt_dst*, *rt_mask*, and *rt_gateway* fields are the route destination address, route address mask, and gateway address, respectively. *rt_mtu* is the maximum transfer unit associated with the route. *rt_flags* contains flags that give some information about a route (for example, whether the route was created dynamically, whether the route is usable, type of route, and so on). *rt_refcnt* indicates the number of references that exist to the route entry. *rt_protocol* indicates how the route entry was generated (for example, configuration, ICMP redirect, and so on). *rt_tos* is the type of service associated with the route. *rt_if* is a NULL-terminated string that represents the interface IP address in dotted decimal format that is associated with the route.

To add a route, the following fields must be set:

- *rt_dst*
- *rt_mask*
- *rt_gateway*
- *rt_tos*
- *rt_protocol*
- *rt_mtu* (Setting the *rt_mtu* value to zero essentially means use the MTU from the associated line description used when the route is bound to an IFC.)
- *rt_if* (*rt_if* can be set to the dotted decimal equivalent of INADDR_ANY, which is 0.)

In addition, the *rt_flags* bit flags can be set to the following:

- RTF_NOEBIND_IFC_FAIL if no rebinding of the route is to occur when the interface associated with the route fails.
- RTF_NOEBIND_IFC_ACTV if no rebinding is to occur when interfaces are activated or

deactivated.

To delete a route, the following fields must be set:

- *rt_dst*
- *rt_mask*
- *rt_gateway*
- *rt_tos*
- *rt_protocol*

All other fields are ignored when adding or removing an entry.

SIOCATMARK Return the value indicating whether socket's read pointer is currently at the out-of-band mark.

The third parameter represents a pointer to an integer flag. If the socket's read pointer is currently at the out-of-band mark, the flag is set to a nonzero value. If it is not, the flag is set to zero.

SIOCDELRT Delete an entry from the interface routing table. Valid for sockets with address family of AF_INET.

See [SIOCADDRT](#) for more information on the third parameter.

SIOCGIFADDR Get the interface address. Valid for sockets with address family of AF_INET.

The third parameter represents a pointer to the structure **ifreq**, defined in `<net/if.h>`:

```
struct ifreq {
    char ifr_name[IFNAMSIZ];
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_mask;
        struct sockaddr ifru_broadaddr;
        short ifru_flags;
        int ifru_mtu;
        int ifru_rbufsize;
        char ifru_linename[10];
        char ifru_TOS;
    } ifr_ifru;
};
```

ifr_name is the name of the interface for which information is to be retrieved. The OS/400 implementation requires this field to be set to a NULL-terminated string that represents the interface IP address in dotted decimal format. Depending on the request, one of the fields in the *ifr_ifru* union will be set upon return from the *ioctl()* call. *ifru_addr* is the local IP address of the interface. *ifru_mask* is the subnetwork mask associated with the interface. *ifru_broadaddr* is the broadcast address. *ifru_flags* contains flags that give some information about an interface (for example, token-ring routing support, whether interface is active, broadcast address, and so on). *ifru_mtu* is the maximum transfer unit configured for the interface. *ifru_rbufsize* is the reassembly buffer size of the interface. *ifru_linename* is the line name associated with the interface. *ifru_TOS* is the type of service configured for the interface.

SIOCGIFBRDADDR Get the interface broadcast address. Valid for sockets with address family of AF_INET.

See [SIOCGIFADDR](#) for more information on the third parameter.

SIOCGIFCONF Get the interface configuration list. Valid for sockets with address family of AF_INET.

The third parameter represents a pointer to the structure **ifconf**, defined in **<net/if.h>**:

```
struct ifconf [
    int ifc_len;
    int ifc_configured;
    int ifc_returned;
    union {
        caddr_t ifcu_buf;
        struct ifreq *ifcu_req;
    } ifc_ifcu;
];
```

ifc_len is a value-result field. The caller passes the size of the buffer pointed to by *ifcu_buf*. On return, *ifc_len* contains the amount of storage that was used in the buffer pointed to by *ifcu_buf* for the interface entries. *ifc_configured* is the number of interface entries in the interface list. *ifc_returned* is the number of interface entries that were returned (this is dependent on the size of the buffer pointed to by *ifcu_buf*). *ifcu_buf* is the user buffer in which a list of interface entries will be stored. Each stored entry will be an *ifreq* structure.

To get the interface configuration list, the following fields must be set:

- *ifc_len*
- *ifcu_buf*

See [SIOCGIFADDR](#) for more information on the list of *ifreq* structures returned. For this request, the *ifr_name* and *ifru_addr* fields will be set to a value.

Note: Additional information about each individual interface can be obtained using these values and the other interface-related requests.

SIOCGIFFLAGS Get interface flags. Valid for sockets with address family of AF_INET.

See [SIOCGIFADDR](#) for more information on the third parameter.

SIOCGIFLIND Get the interface line description name. Valid for sockets with address family of AF_INET.

See [SIOCGIFADDR](#) for more information on the third parameter.

SIOCGIFMTU Get the interface network MTU. Valid for sockets with address family of AF_INET.

See [SIOCGIFADDR](#) for more information on the third parameter.

SIOCGIFNETMASK Get the mask for the network portion of the interface address. Valid for sockets with address family of AF_INET.

See [SIOCGIFADDR](#) for more information on the third parameter.

SIOCGIFRBUFS Get the interface reassembly buffer size. Valid for sockets with address family of AF_INET.

See [SIOCGIFADDR](#) for more information on the third parameter.

SIOCGIFTOS Get the interface type-of-service (TOS). Valid for sockets with address family of AF_INET.

See [SIOCGIFADDR](#) for more information on the third parameter.

SIOCGPGRP Get the process ID or process group ID that is to receive the SIGIO and SIGURG signals.

See [FIOGETOWN](#) for more information on the third parameter.

SIOCGRTCONF

Get the route configuration list. Valid for sockets with address family of AF_INET.

For the SIOCGRTCONF request, the third parameter represents a pointer to the structure **rtconf**, also defined in **<net/route.h>**:

```
struct rtconf [
    int rtc_len;
    int rtc_configured;
    int rtc_returned;
    union {
        caddr_t rtcu_buf;
        struct rtenry *rtcu_req;
    } rtc_rtcu;
];
```

rtc_len is a value-result field. The caller passes the size of the buffer pointed to by *rtcu_buf*. On return, *rtc_len* contains the amount of storage that was used in the buffer pointed to by *rtcu_buf* for the route entries. *rtc_configured* is the number of route entries in the route list. *rtc_returned* is the number of route entries that were returned (this is dependent on the size of the buffer pointed to by *rtcu_buf*). *rtcu_buf* is the user buffer in which a list of route entries will be stored. Each stored entry will be an *rtenry* structure.

To get the route configuration list, the following fields must be set:

- *rtc_len*
- *rtcu_buf*

See [SIOCADDRRT](#) for more information on the list of *rtenry* structures returned. For this request, all fields in each *rtenry* structure will be set to a value.

SIOCSNDQ

Return the number of bytes on the send queue that have not been acknowledged by the remote system. Valid for sockets with address family of AF_INET or AF_INET6 and socket type of SOCK_STREAM.

The third parameter represents a pointer to an integer that is set to the number of bytes yet to be acknowledged as being received by the remote TCP transport driver.

Notes:

1. SIOCSNDQ is used after a series of blocking or non-blocking send operations to see if the sent data has reached the transport layer on the remote system. Note that this does not guarantee the data has reached the remote application.
2. When SIOCSNDQ is used in a multithreaded application, the actions of other threads must be considered by the application. SIOCSNDQ provides a result for a socket descriptor at the given point in time when the *ioctl()* request is received by the TCP transport layer. Blocking send operations that have not completed, as well as non-blocking send operations in other threads issued after the SIOCSNDQ *ioctl()*, are not reflected in the result obtained for the SIOCSNDQ *ioctl()*.
3. In a situation where the application has multiple threads sending data on the same socket descriptor, the application should not assume that all data has been received by the remote side when 0 is returned if the application is not positive that all send operations in the other threads were complete at the time the SIOCSNDQ *ioctl()* was issued. An application should issue the SIOCSNDQ *ioctl()* only after it has completed all of the send operations. No value is added by querying the machine to see if it has sent all of the data when the application itself has not sent all of the data in a given unit of work.

SIOCSGRP

Set the process ID or process group ID that is to receive the SIGIO and SIGURG signals.

See [FIOSETOWN](#) for more information on the third parameter.

SIOCSTELRSC

Set telephony resources. Valid for sockets with address family of AF_TELEPHONY.

The third parameter represents a pointer to a TelResource structure, which is defined in `<nettel/tel.h>`.

```
struct TelResource {                                /* telephony resource structure */
    int    trCount;                                /* number of devices           */
    char   trReserved[12];                          /* reserved                     */
    void*  trResourceList;                          /* pointer to array of system   */
                                                /* pointers                      */
};
```

trCount is the number of devices that are to be associated with the socket, *trReserved* is a reserved field, and *trResourceList* is a pointer to an array of space pointers. Each of these space pointers is the address of a device that will be associated with the socket.

Notes:

1. This request will associate one or more telephony (*TEL) network devices with a socket. Once the association is made, it will last until the socket is closed.
2. The user is responsible for resolving each device name to a system pointer.
3. Before the device can be associated with a socket, the following conditions must be met:
 - The PPP line, network controller, and network device descriptions must exist.
 - The PPP line must be associated with an ISDN network controller.
 - The PPP line must be associated with a connection list and connection list entry (inbound or outbound, as appropriate).
 - The line, controller, and device must be varied on.
4. The user must have at least operational authority for the devices to be associated with the socket.
5. A device cannot be associated with more than one socket at a time.
6. If the SIOCSTELRSC request fails for any reason, none of the specified devices will be associated the socket.
7. For more information about this request and the AF_TELEPHONY address family, please see [Socket address family](#).

Authorities

No authorization is required.

Return Value

ioctl() returns an integer. Possible values are:

- 0 (*ioctl()* was successful)
- -1 (*ioctl()* was not successful. The *errno* global variable is set to indicate the error.)

Error Conditions

If *ioctl()* is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES] Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN] Operation would have caused the process to be suspended.

[EBADF] Descriptor not valid.

A descriptor argument was out of range, referred to an object that was not open, or a read or write request was made to an object that is not open for that operation.

A given descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open object.

[EBADFID] A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBUSY] Resource busy.

An attempt was made to use a system resource that is not available at this time.

[EDAMAGE] A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFAULT] The address used for an argument is not correct.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

»*[EINTR]* Interrupted function call.«

[EINVAL] The value specified for an argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL. Either the requested function is not supported, or the optional parameter is not valid.

[EIO] Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[ENOBUFFS] There is not enough buffer space for the requested operation.

[ENOSPC] No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended object.

- [*ENOSYS*] Function not implemented.
- An attempt was made to use a function that is not available in this implementation for any object or any arguments.
- The path name given refers to an object that does not support this function.
- [*ENOTAVAIL*] Independent Auxiliary Storage Pool (ASP) is not available.
- The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.
- To recover from this error, wait until processing has completed for the independent ASP.
- [*ENOTSAFE*] Function is not allowed in a job that is running with multiple threads.
- [*EPERM*] Operation not permitted.
- You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.
- [*EPIPE*] Broken pipe.
- »[*ERESTART*] A system call was interrupted and may be restarted.«
- [*ESTALE*] File or object handle rejected by server.
- If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
- [*EUNATCH*] The protocol required to support the specified address family is not available at this time.
- [*EUNKNOWN*] Unknown system state.
- The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

- [*EADDRNOTAVAIL*] Address not available.
- [*ECONNABORTED*] Connection ended abnormally.
- [*ECONNREFUSED*] The destination socket refused an attempted connect operation.
- [*ECONNRESET*] A connection with a remote socket was reset by that socket.
- [*EHOSTDOWN*] A remote host is not available.
- [*EHOSTUNREACH*] A route to the remote host is not available.
- [*ENETDOWN*] The network is not currently available.
- [*ENETRESET*] A socket is connected to a host that is no longer available.
- [*ENETUNREACH*] Cannot reach the destination network.
- [*ETIMEDOUT*] A remote host did not respond within the timeout period.

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPFA0D4 E	File system error occurred. Error number &1.
CPFA081 E	Unable to set return value or error code.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - **»Independent ASP QSYS.LIB«**
 - QOPT
2. QDLS File System Differences

QDLS does not support **ioctl()**.
3. QOPT File System Differences

QOPT does not support **ioctl()**.
4. A program must have the appropriate privilege *IOSYSCFG to issue any of the following requests: SIOCADDRT and SIOCDELRT.

Related Information

- The `<sys/ioctl.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<sys/types.h>` file (see [Header Files for UNIX-Type Functions](#))
- [fcntl\(\)](#)--Perform File Control Command
- [Socket Programming](#)

lchown()--Change Owner and Group of Symbolic Link

Syntax

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

```
int lchown(const char *path, uid_t owner, gid_t group);
```

Threadsafe: Conditional; see Usage Notes.

The **lchown()** function changes the owner and group of a file. If the named file is a symbolic link, **lchown()** changes the owner or group of the link itself rather than the object to which the link points. The permissions of the previous owner or primary group to the object are revoked.

If the file is checked out by another user (someone other than the user profile of the current job), **lchown()** fails with the [EBUSY] error.

When **lchown()** completes successfully, it updates the change time of the file.

Parameters

path

(Input) A pointer to the null-terminated path name of the file whose owner and group are being changed.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See [QlgLchown\(\)--Change Owner and Group of Symbolic Link](#) for a description and an example of supplying the *path* in any CCSID.

owner

(Input) The user ID (uid) of the new owner of the file. If the value is -1, the user ID is not changed.

group

(Input) The group ID (gid) of the new group for the file. If the value is -1, the group ID is not changed.

Note: Changing the owner or the primary group causes the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode to be cleared, unless the caller has *ALLOBJ special authority. If the caller does have *ALLOBJ special authority the bits are not changed. This does not apply to directories. See the **chmod()** documentation.

Authorities

Note: Adopted authority is not used.

Authorization Required for lchown() (excluding QSYS.LIB, independent ASP QSYS.LIB, and QDLS)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object, when changing the owner	Owner and *OBJEXIST (also see Note 1)	EPERM
Object, when changing the primary group	See Note 2	EPERM
Previous owner's user profile, when changing the owner	*DLT	EPERM
New owner's user profile, when changing the owner	*ADD	EPERM
User profile of previous primary group, when changing the primary group	*DLT	EPERM
New primary group's user profile, when changing the primary group	*ADD	EPERM


Note:

1. You do not need the listed authority if you have *ALLOBJ special authority.
2. At least one of the following must be true:
 - a. You have *ALLOBJ special authority.
 - b. You are the owner and either of the following:
 - The new primary group is the primary group of the job.
 - The new primary group is one of the supplementary groups of the job.

Authorization Required for lchown() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X See Note 1	EACCES
Object when changing the owner	See Note 2(a)	EPERM
Object when changing the primary group	See Note 2(b)	EPERM

Note:

1. For *FILE objects (such as DDM file, diskette file, print file, and save file), *RX authority is required to the parent directory of the object, rather than just *X authority.
2. The required authorization varies for each object type. For details of the following commands see the [iSeries Security Reference](#)  book.
 - a. CHGOWN
 - b. CHGPGP

Authorization Required for lchown() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object	*ALLOBJ Special Authority or Owner	EPERM
Previous owner's user profile, when changing the owner	*DLT	EPERM
New owner's user profile, when changing the owner	*ADD	EPERM
Previous primary group's user profile, when changing the primary group	*DLT	EPERM
New primary group's user profile, when changing the primary group	*ADD	EPERM

Authorization Required for `lchown()` in the QOPT File System

Object Referred to	Authority Required	errno
Volume authorization list	*CHANGE	EACCES
Each directory in the path name preceding the object.	*X	EACCES
Object	*ALLOBJ Special Authority or Owner	EPERM

Return Value

0

`lchown()` was successful.

-1

`lchown()` was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If `lchown()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

owner or *group* is not a valid user ID (uid) or group ID (gid).

owner is the current primary group of the object.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

»[EJRNDAMAGE]

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.

[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal. ❄

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than POSIX_SYMLoop (defined in the limits.h header file). Symbolic links are encountered during resolution of the directory or path name.

[ENAMETOOLONG]

A path name is too long.

A path name is longer than PATH_MAX characters or some component of the name is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length

of the name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using the **pathconf()** function.

»[ENEWJRN]

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal.«

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[EPERM]

Operation not permitted.

You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.

The thread does not have authority to perform the requested function.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

Error Messages

The following messages may be sent from this function:

CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC

- QSYS.LIB
- »Independent ASP QSYS.LIB «
- QOPT

2. QSYS.LIB »and Independent ASP QSYS.LIB «File System Differences

lchown() is not supported for member (.MBR) objects.

3. QDLS File System Differences

The owner and primary group of the /QDLS directory (root folder) cannot be changed. If an attempt is made to change the owner and primary group, a [ENOTSUP] error is returned.

4. QOPT File System Differences

Changing the owner and primary group is allowed only for an object that exists on a volume formatted in Universal Disk Format (UDF). For all other media formats, ENOTSUP will be returned.

QOPT file system objects that have owners will not be recognized by the Work with Objects by Owner (WRKOBJOWN) CL command. Likewise, QOPT objects that have a primary group will not be recognized by the Work Objects by Primary Group (WRKOBJPGP) CL command.

5. QFileSvr.400 File System Differences

The QFileSvr.400 file system does not support **lchown()**.

6. QNetWare File System Differences

The QNetWare file system does not support primary group. The GID must be zero.

7. QNTC File System Differences

The owner of files and directories cannot be changed. All files and directories in QNTC are owned by the QDFTOWN user profile.

Related Information

- The <**unistd.h**> file (see [Header Files for UNIX-Type APIs](#))
- The <**limits.h**> file
- [chmod\(\)--Change File Authorizations](#)
- [fchown\(\)--Change Owner and Group of File by Descriptor](#)
- [fstat\(\)--Get File Information by Descriptor](#)
- [lstat\(\)--Get File or Link Information](#)
- [stat\(\)--Get File Information](#)

- [OlgLchown\(\)--Change Owner and Group of Symbolic Link](#)

Example

The following example changes the owner and group of a file:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

main() {
    char link_name[]="temp.link";
    char fn[]="temp.file";
    struct stat info;

    if (symlink(fn, link_name) == -1)
        perror("symlink() error");
    else {
        lstat(link_name, &info);
        printf("original owner was %d and group was %d\n", info.st_uid,
            info.st_gid);
        if (lchown(link_name, 152, 0) != 0)
            perror("lchown() error");
        else {
            lstat(link_name, &info);
            printf("after lchown(), owner is %d and group is %d\n",
                info.st_uid, info.st_gid);
        }
        unlink(link_name);
    }
}
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

link()--Create Link to File

Syntax

```
#include <unistd.h>

int link(const char *existing, const char *new);
```

Threadsafe: Conditional; see Usage Notes.

The **link()** function provides an alternative path name for the existing file, so that the file can be accessed by either the existing name or the new name. **link()** creates a link with a path name *new* to an existing file whose path name is *existing*. The link can be stored in the same directory as the original file or in a different directory.

The **link()** function creates a hard link, which guarantees the existence of a file even after the original path name has been removed.

If **link()** successfully creates the link, it increments the *link count* of the file. The link count indicates how many links there are to the file. If **link()** fails for some reason, the link count is not incremented.

If the *existing* argument names a symbolic link, **link()** creates a link that refers to the file that results from resolving the path name contained in the symbolic link. If *new* names a symbolic link, **link()** fails and sets *errno* to [EEXIST].

A successful link updates the change time of the file, and the change time and modification time of the directory that contains *new* (parent directory).

If the file is checked out by another user (a user profile other than the user profile of the current job), **link()** fails with the [EBUSY] error.

Links created by this function are not allowed to cross file systems. For example, you cannot create a link to a file in the QOpenSys directory from the root (/) directory.

Links are not allowed to directories. If *existing* names a directory, **link()** fails and sets *errno* to [EPERM].

A job must have access to a file to link to it.

Parameters

existing

(Input) A pointer to a null-terminated path name naming an existing file to which a new link is to be created.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See [QlgLink\(\)--Create Link to File](#) for a description and an example of supplying the *existing* in any CCSID.

new

(Input) A pointer to a null-terminated path name that is the name of the new link.

This parameter is assumed to be represented in the CCSID currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job. The new link name is assumed to be represented in the language and country or region currently in effect for the job.

See [QlgLink\(\)--Create Link to File](#) for a description and an example of supplying the *new* in any CCSID.

Authorities

Note: Adopted authority is not used.

Authorization Required for link()

Object Referred to	Authority Required	errno
Each directory in the <i>existing</i> path name that precedes the object being linked to	*X	EACCES
<i>Existing</i> object	*OBJEXIST	EACCES
Each directory in the <i>new</i> path name that precedes the object being linked to	*X	EACCES
Parent directory of the new link	*WX	EACCES

Return Value

0

link() was successful.

-1

link() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **link()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by

the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EEXIST]

File exists.

The file specified already exists and the specified operation requires that it not exist.

The named file, directory, or path already exists.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[EISDIR]

Specified target is a directory.

The path specified named a directory where a file or object name was expected.

The path name given is a directory.

➤ *[EJRNDAMAGE]*

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNENTTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.

[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been

exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal. <<

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than POSIX_SYMLoop (defined in the limits.h header file). Symbolic links are encountered during resolution of the directory or path name.

[EMLINK]

Maximum link count for a file was exceeded.

An attempt was made to have the link count of a single file exceed LINK_MAX. The value of LINK_MAX can be determined using the pathconf() or the fpathconf() function.

[ENAMETOOLONG]

A path name is too long.

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

>>[ENEWJRN]

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal. <<

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOSYS]

Function not implemented.

An attempt was made to use a function that is not available in this implementation for any object or any arguments.

The path name given refers to an object that does not support this function.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[EPERM]

Operation not permitted.

You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.

Links to directories are not supported.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

[EXDEV]

Improper link.

A link to a file on another file system was attempted.

Error Messages

The following messages may be sent from this function:

- CPE3418 E Possible APAR condition or hardware failure.
- CPFA0D4 E File system error occurred. Error number &1.
- CPF3CF2 E Error(s) occurred during running of &1 API.
- CPF9872 E Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined

- QNTC
- QSYS.LIB
- [»Independent ASP QSYS.LIB«](#)
- QOPT

2. The **link()** function should be used sparingly to avoid potential performance degradation. The greater the number of hard links to an object, the more time it will take to change the attributes of the object.

3. File System Differences

The following file systems do not support **link()**:

- QSYS.LIB
- [»Independent ASP QSYS.LIB«](#)
- QDLS
- QOPT
- QFileSvr.400
- QNetWare
- QNTC

If **link()** is used in any of these file systems, a [ENOSYS] error is returned.

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))
- [QlgLink\(\)--Create Link to File](#)
- [rename\(\)--Rename File or Directory](#)
- [unlink\(\)--Remove Link to File](#)

Example

The following example uses **link()**:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

main()
{
    char fn[]="link.example.file";
    char ln[]="link.example.link";
```

```

int file_descriptor;
struct stat info;

if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
else {
    close(file_descriptor);
    puts("before link()");
    stat(fn,&info);
    printf("    number of links is %hu\n",info.st_nlink);
    if (link(fn, ln) != 0) {
        perror("link() error");
        unlink(fn);
    }
    else {
        puts("after link()");
        stat(fn,&info);
        printf("    number of links is %hu\n",info.st_nlink);
        unlink(ln);
        puts("after first unlink()");
        stat(fn,&info);
        printf("    number of links is %hu\n",info.st_nlink);
        unlink(fn);
    }
}
}
}

```

Output:

```

before link()
    number of links is 1
after link()
    number of links is 2
after first unlink()
    number of links is 1

```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

lseek()--Set File Read/Write Offset

Syntax

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

```
off_t lseek(int file_descriptor, off_t offset, int whence);
```

Threadsafe: Conditional; see Usage Notes.

The **lseek()** function changes the current file offset to a new position in the file. The new position is the given byte *offset* from the position specified by *whence*. After you have used **lseek()** to seek to a new location, the next I/O operation on the file begins at that location.

lseek() lets you specify new file offsets past the current end of the file. If data is written at such a point, read operations in the gap between this data and the old end of the file will return bytes containing binary zeros (or bytes containing blanks in the QSYS.LIB and independent ASP QSYS.LIB file systems). In other words, the gap is assumed to be filled with zeros (or with blanks in the QSYS.LIB and independent ASP QSYS.LIB file systems). Seeking past the end of a file, however, does not automatically extend the length of the file. There must be a write operation before the file is actually extended.

There are some important considerations for **lseek()** if the O_TEXTDATA and O_CCSID flags were specified on the **open()**, the file CCSID and open CCSID are not the same, and the converted data could expand or contract:

- Making assumptions about data size and the current file offset is extremely dangerous. For example, a file might have a physical size of 100 bytes, but after an application has read 100 bytes from the file, the current file offset may be only 50. To read the whole file, the application might have to read 200 bytes or more, depending on the CCSIDs involved. Therefore, **lseek()** will only be allowed to change the current file offset to:
 - The start of the file (*offset 0, whence SEEK_SET*)
 - The end of the file (*offset 0, whence SEEK_END*). In this case, the function will return a calculated value based on the physical size of the file, the CCSID of the file, and the CCSID of the open instance. This may be different than the actual file offset.

If any other combination of values is specified, **lseek()** fails and *errno* is set to ENOTSUP.

- Internally-buffered data from a read or write operation is discarded. See [read\(\)--Read from Descriptor](#) and [write\(\)--Write to Descriptor](#) for more information concerning internal buffering of text data.
- The expected state for the current text conversion is reset to the initial state. This consideration applies only when using a CCSID that can represent data using more than one graphic character set or containing characters of different byte lengths. Some CCSIDs require an escape or shift sequence to signify a state change from one character set or byte length to another. Failing to account for this consideration could lead to incorrect text conversion if, for instance, a double-byte character at the new file offset was treated as two single-byte characters by the conversion function.

In the QSYS.LIB file and independent ASP QSYS.LIB file systems, you can seek only to the beginning of a member while in text mode.

Parameters

file_descriptor

(Input) The file whose current file offset you want to change.

offset

(input) The amount (positive or negative) the byte offset is to be changed. The sign indicates whether the offset is to be moved forward (positive) or backward (negative).

whence

(Input) One of the following symbols (defined in the `<unistd.h>` header file):

SEEK_SET

The start of the file

SEEK_CUR

The current file offset in the file

SEEK_END

The end of the file

If bits in *whence* are set to values other than those defined above, **lseek()** fails with the [EINVAL] error.

Authorities

No authorization is required. Authorization is verified during **open()** or **creat()**.

Return Value

value

lseek() was successful. The value returned is the new file offset, measured in bytes from the beginning of the file.

-1

lseek() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **lseek()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADF]

Descriptor not valid.

A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL. A parameter passed to this function is not valid.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOSYSRSC]

System resources not available to complete request.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[EOVERFLOW]

Object is too large to process.

The object's data size exceeds the limit allowed by this function.

The resulting file offset would be a value that cannot be represented correctly in a variable of type `off_t` (the offset is greater than 2GB minus 2 bytes).

[ESPIPE]

Seek request not supported for object.

A seek request was specified for an object that does not support seeking.

The object is not capable of seeking.

The *file_descriptor* argument is associated with a pipe or FIFO.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages



The following messages may be sent from this function:

CPE3418 E Possible APAR condition or hardware failure.
CPFA0D4 E File system error occurred. Error number &1.
CPF3CF2 E Error(s) occurred during running of &1 API.
CPF9872 E Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

- Where multiple threads exist in the job.
- The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

- Root
- QOpenSys
- User-defined
- QNTC
- QSYS.LIB
- Independent ASP QSYS.LIB 
- QOPT

2. Network File System Differences

Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations (several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data).

3. QSYS.LIB  and Independent ASP QSYS.LIB  File System Differences

This function is not supported for save files and will fail with error code [ENOTSUP].

4. This function will fail with the [E_OVERFLOW] error if the resulting file offset would be a value that cannot be represented correctly in a variable of type off_t (the offset is greater than 2 GB minus 2 bytes).
5. When you develop in C-based languages and an application is compiled with the `_LARGE_FILES` macro defined, the `lseek()` API will be mapped to a call to the `lseek64()` API. Additionally, the data type `off_t` will be mapped to the type `off64_t`.
6. Using this function with the `write()`, `⌘pwrite()`, and `pwrite64()` functions on the `/dev/null` or `/dev/zero` `⌘`character special file will not result in the file data size changing from zero.

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))
- [creat\(\)--Create or Rewrite File](#)
- [dup\(\)--Duplicate Open File Descriptor](#)
- [fcntl\(\)--Perform File Control Command](#)
- [lseek64\(\)--Set File Read/Write Offset \(Large File Enabled\)](#)
- [open\(\)--Open File](#)
- [⌘pread\(\)--Read from Descriptor with Offset](#) `⌘`
- [⌘pread64\(\)--Read from Descriptor with Offset \(large file enabled\)](#) `⌘`
- [⌘pwrite\(\)--Write to Descriptor with Offset](#) `⌘`
- [⌘pwrite64\(\)--Write to Descriptor with Offset \(large file enabled\)](#) `⌘`
- [read\(\)--Read from Descriptor](#)
- [write\(\)--Write to Descriptor](#)

Example

The following example positions a file (that has at least 11 bytes) to an offset of 10 bytes before the end of the file:

```
lseek(file_descriptor, -10, SEEK_END);
```

API introduced: V4R4

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

lseek64()--Set File Read/Write Offset (Large File Enabled)

Syntax

```
#include <unistd.h>

off64_t lseek64(int file_descriptor,
                off64_t offset, int whence);
```

Threadsafe: Conditional; see Usage Notes.

The **lseek64()** function changes the current file offset to a new position in the file. The new position is the given byte *offset* from the position specified by *whence*. After you have used **lseek64()** to seek to a new location, the next I/O operation on the file begins at that location.

lseek64() lets you specify new file offsets past the current end of the file. If data is written at such a point, read operations in the gap between this data and the old end of the file will return bytes containing binary zeros (or bytes containing blanks in the QSYS.LIB or independent ASP QSYS.LIB file systems). In other words, the gap is assumed to be filled with zeros (or with blanks in the QSYS.LIB or independent ASP QSYS.LIB file systems). If you seek past the end of a file, however, the length of the file is not automatically extended. The maximum offset that can be specified is the largest value that can be held in an 8-byte, signed integer. You must do a write operation before the file is actually extended.

In the QSYS.LIB or independent ASP QSYS.LIB file systems, you can seek only to the beginning of a member while in text mode.

lseek64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 byte as long as the file has been opened by either of the following:

- Using the **open64()** function (see [open64\(\)--Open File \(Large File Enabled\)](#)).
- Using the **open()** function (see [open\(\)--Open File](#)) with the O_LARGEFILE flag set.

For additional information about parameters, authorities required, error conditions and examples, see [lseek\(\)--Set File Read/Write Offset](#).

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **lseek64()** API and the off64_t data type, you must compile the source with the `_LARGE_FILE_API` defined.
2. All of the usage notes for **lseek()** apply to **lseek64()**. See [Usage Notes](#) in the **lseek()** API.

API introduced: V4R4

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

Istat()--Get File or Link Information

Syntax

```
#include <sys/stat.h>
```

```
int lstat(const char *path, struct stat *buf);
```

Threadsafe: Conditional; see Usage Notes.

The **lstat()** function gets status information about a specified file and places it in the area of memory pointed to by *buf*. If the named file is a symbolic link, **lstat()** returns information about the symbolic link itself.

The information is returned in the stat structure, referenced by *buf*. For details on the stat structure, see [stat\(\)--Get File Information](#).

If the named file is not a symbolic link, **lstat()** updates the time-related fields before putting information in the stat structure.

Parameters

path

(Input) A pointer to the null-terminated path name of the file.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See [QlgLstat\(\)--Get File or Link Information](#) for a description and an example of supplying the *path* in any CCSID.

buf

(Output) A pointer to the area to which the information should be written.

Authorities

Note: Adopted authority is not used.

Authorization Required for lstat()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object	*R	EACCES

Return Value

0

lstat() was successful. The information is returned in *buf*.

-1

lstat() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **lstat()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than POSIX_SYMLOOP (defined in the limits.h header file). Symbolic links are encountered during resolution of the directory or path name.

[ENAMETOOLONG]

A path name is too long.

A path name is longer than PATH_MAX characters or some component of the name is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length

of the name string substituted for a symbolic link exceeds `PATH_MAX`. The `PATH_MAX` and `NAME_MAX` values can be determined using the `pathconf()` function.

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the

requested arguments.

[Eoverflow]

Object is too large to process.

The object's data size exceeds the limit allowed by this function.

The file size in bytes cannot be represented correctly in the structure pointed to by buf (the file is larger than 2GB minus 1 byte).

[Eperm]

Operation not permitted.

You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.

[EROobj]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[Eunknown]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:

- Where multiple threads exist in the job.
- The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - >>Independent ASP QSYS.LIB <<
 - QOPT

2. QOPT File System Differences

The value for `st_atime` will always be zero. The value for `st_ctime` will always be the creation date and time of the file or directory.

If the object exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for the object and each directory in the path name follows the rules described in [Authorization Required for `lstat\(\)`](#). If the object exists on a volume formatted in some other media format, no authorization checks are made on the object or each directory in the path name. The volume authorization list is checked for *USE authority regardless of the volume media format.

The user, group, and other mode bits are always on for an object that exists on a volume not formatted in Universal Disk format (UDF).

`lstat` on `/QOPT` will always return 2,147,483,647 for size fields.

`lstat` on optical volumes will return the volume capacity or 2,147,483,647, whichever is smaller.

The file access time is not changed.

3. Network File System Differences

Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)

4. QNetWare File System Differences

The QNetWare file system does not fully support mode bits. See [NetWare on iSeries](#) for more information.

5. This function will fail with the [E_OVERFLOW] error if the file size in bytes cannot be represented correctly in the structure pointed to by `buf` (the file is larger than 2GB minus 1 byte).
6. When you develop in C-based languages and this function is compiled with `_LARGE_FILES` defined, it will be mapped to `lstat64()`. Note that the type of the `buf` parameter, `struct stat`, also will be mapped to type `struct stat64`.

Related Information

- The `<sys/stat.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<sys/types.h>` file (see [Header Files for UNIX-Type Functions](#))
- [chmod\(\)--Change File Authorizations](#)
- [chown\(\)--Change Owner and Group of File](#)

- [creat\(\)--Create or Rewrite File](#)
- [dup\(\)--Duplicate Open File Descriptor](#)
- [fcntl\(\)--Perform File Control Command](#)
- [fstat\(\)--Get File Information by Descriptor](#)
- [link\(\)--Create Link to File](#)
- [mkdir\(\)--Make Directory](#)
- [open\(\)--Open File](#)
- [Qlglstat\(\)--Get File or Link Information](#)
- [read\(\)--Read from Descriptor](#)
- [readlink\(\)--Read Value of Symbolic Link](#)
- [stat\(\)--Get File Information](#)
- [symlink\(\)--Make Symbolic Link](#)
- [unlink\(\)--Remove Link to File](#)
- [utime\(\)--Set File Access and Modification Times](#)
- [write\(\)--Write to Descriptor](#)

Example

The following example provides status information for a file:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>

main() {
    char fn[]="temp.file", ln[]="temp.link";
    struct stat info;
    int file_descriptor;

    if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(file_descriptor);
        if (link(fn, ln) != 0)
            perror("link() error");
        else {
            if (lstat(ln, &info) != 0)
                perror("lstat() error");
            else {
                puts("lstat() returned:");
                printf("  inode:   %d\n",    (int) info.st_ino);
            }
        }
    }
}
```

```
        printf(" dev id:   %d\n",    (int) info.st_dev);
        printf("   mode:   %08x\n",   info.st_mode);
        printf("  links:   %d\n",     info.st_nlink);
        printf("   uid:    %d\n",    (int) info.st_uid);
        printf("   gid:    %d\n",    (int) info.st_gid);
    }
    unlink(ln);
}
    unlink(fn);
}
}
```

Output:

```
lstat() returned:
inode:   3022
dev id:  1
mode:    00008080
links:   2
uid:     137
gid:     500
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

Istat64()--Get File or Link Information (Large File Enabled)

Syntax

```
#include <sys/stat.h>

int lstat64(const char *path, struct stat64 *buf);
```

Threadsafe: Conditional; see Usage Notes.

The **lstat64()** function gets status information about a specified file and places it in the area of memory pointed to by *buf*. If the named file is a symbolic link, **lstat64()** returns information about the symbolic link itself.

The information is returned in the `stat64` structure, referred to by *buf*. For details on the `stat64` structure, see [stat64\(\)--Get File Information \(Large File Enabled\)](#).

If the named file is not a symbolic link, **lstat64()** updates the time-related fields before putting information in the `stat64` structure.

For additional information about parameters, authorities required, and error conditions, see [lstat\(\)--Get File or Link Information](#).

See [QlgLstat64\(\)--Get File or Link Information \(Large File Enabled\)](#) for a description and an example of supplying the *path* in any CCSID.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **lstat64()** API and the `struct stat64` data type, you must compile the source with the `_LARGE_FILE_API` defined.
2. All of the usage notes for **lstat()** apply to **lstat64()**. See [Usage Notes](#) in the **lstat()** API.

Example

The following example provides status information for a file.

```
#define _LARGE_FILE_API
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
```

```

#include <stdio.h>

main() {
    char fn[]="temp.file", ln[]="temp.link";
    struct stat64 info;
    int file_descriptor;

    if ((file_descriptor = creat64(fn, S_IWUSR)) < 0)
        perror("creat64() error");
    else {
        close(file_descriptor);
        if (link(fn, ln) != 0)
            perror("link() error");
        else {
            if (lstat64(ln, &info) != 0)
                perror("lstat64() error");
            else {
                puts("lstat64() returned:");
                printf("  inode:   %d\n", (int) info.st_ino);
                printf(" dev id:   %d\n", (int) info.st_dev);
                printf("  mode:   %08x\n", info.st_mode);
                printf("  links:  %d\n", info.st_nlink);
                printf("   uid:   %d\n", (int) info.st_uid);
                printf("   gid:   %d\n", (int) info.st_gid);
                printf("  size:  %lld\n", (long long) info.st_size);
            }
            unlink(ln);
        }
        unlink(fn);
    }
}

```

Output:

```

lstat() returned:
  inode:   3022
 dev id:   1
  mode:   00008080
  links:  2
   uid:   137
   gid:   500
  size:   18

```

mkdir()--Make Directory

Syntax

```
#include <sys/stat.h>

int mkdir(const char *path, mode_t mode);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **mkdir()** function creates a new, empty directory whose name is defined by *path*. The file permission bits in *mode* are modified by the file creation mask of the job and then used to set the file permission bits of the directory being created.

For more information on the permission bits in *mode* see [chmod\(\)--Change File Authorizations](#). For more information on the file creation mask, see [umask\(\)--Set Authorization Mask for Job](#).

The owner ID of the new directory is set to the effective user ID (uid) of the job. **»**If the directory is being created in the Root (/), QOpensys, and user-defined file systems, the following applies. If the S_ISGID bit of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the directory. If the S_ISGID bit of the parent directory is on, the group ID (GID) of the new directory is set to the GID of the parent directory. For all other file systems, the **«**group ID (GID) of the new directory is set to the GID of the parent directory.

mkdir() sets the access, change, modification, and creation times for the new directory. It also sets the change and modification times for the directory that contains the new directory (parent directory).

The link count of the parent directory link count is increased by one. The link count of the new directory is set to 2. The new directory also contains an entry for "dot" (.) and "dot-dot" (..).

If *path* names a symbolic link, the symbolic link is not followed, and **mkdir()** fails with the [EEXIST] error.

If bits in *mode* other than the file permission bits are set, **mkdir()** fails with the [EINVAL] error.

Parameters

path

(Input) A pointer to the null-terminated path name of the directory to be created.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

The name of the new directory is assumed to be represented in the language and country or region currently in effect for the process.

See [QlgMkdir\(\)--Make Directory](#) for a description and an example of supplying the *path* in any CCSID.

mode

(Input) Permission bits for the new directory. [»](#)The S_ISGID (set-group-ID) bit also may be specified when creating the directory.

See [chmod\(\)--Change File Authorizations](#) for details on the values that can be specified for *mode*.
[«](#)

Authorities

Note: Adopted authority is not used.

Authorization Required for mkdir() (excluding QSYS.LIB, [»](#)Independent ASP QSYS.LIB, [«](#)and QDLS)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the directory to be created.	*X	EACCES
Parent directory of directory to be created	*WX	EACCES

Authorization Required for mkdir() in the QSYS.LIB [»](#)and independent ASP QSYS.LIB File Systems [«](#)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the directory to be created.	*X	EACCES
Parent directory of directory to be created (when the directory being created is a database file)	*X and *ADD	EACCES

Authorization Required for mkdir() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the directory to be created.	*X	EACCES
Parent directory of directory to be created	*CHANGE	EACCES

Return Value

0

mkdir() was successful. The directory was created.

-1

mkdir() was not successful. The directory was not created. The *errno* global variable is set to indicate the error.

Error Conditions

If `mkdir()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EEXIST]

File exists.

The file specified already exists and the specified operation requires that it not exist.

The named file, directory, or path already exists. Or, the last component of *path* is a symbolic link.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

➤ *[EJRNDAMAGE]*

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNTTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.

[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal. ❄

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than POSIX_SYMLoop (defined in the limits.h header file). Symbolic links are encountered during resolution of the directory or path name.

[EMLINK]

Maximum link count for a file was exceeded.

An attempt was made to have the link count of a single file exceed LINK_MAX. The value of LINK_MAX can be determined using the pathconf() or the fpathconf() function.

[ENAMETOOLONG]

A path name is too long.

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

❄*[ENEWJRN]*

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal. ❄

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOSYS]

Function not implemented.

An attempt was made to use a function that is not available in this implementation for any object or any arguments.

The path name given refers to an object that does not support this function.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[EPERM]

Operation not permitted.

You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]



The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:


CPE3418 E Possible APAR condition or hardware failure.
CPFA0D4 E File system error occurred. Error number &1.
CPF3CF2 E Error(s) occurred during running of &1 API.
CPF9872 E Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - There are secondary threads active in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - Independent ASP QSYS.LIB 
 - QOPT

2. Root, QOpenSys, and User-Defined File System Differences

The user who creates the directory becomes its owner.

The S_ISGID bit of the directory affects what the group ID (GID) is for objects that are created in the directory. If the S_ISGID bit of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the object. If the S_ISGID bit of the parent directory is on, the

group ID (GID) is copied from the parent directory in which the new directory is being created.

The owner, primary group, and public object authorities (*OBJEXIST, *OBJMGT, *OBJALTER, and *OBJREF) are copied from the parent directory's owner, primary group, and public object authorities. This occurs even when the new directory has a different owner than the parent directory. The owner, primary group, and public data authorities (*R, *W, and *X) are derived from the permissions specified in the mode (except for those permissions that are also set in the file mode creation mask). The new directory does not have any private authorities or authorization list. It only has authorities for the owner, primary group, and public.

3. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

The user who creates the directory becomes its owner. The group ID is copied from the primary user ID, if one exists.

The owner is given *ALL object authority to the new directory. The group object authorities are copied from the user profile of the owner. The public receives no object authority to the directory.

The primary group authorities specified in *mode* are not saved if no primary group exists.

The change and modification times for the directory that contains the new directory are only set when the new directory is a database file.

4. QDLS File System Differences

The user who creates the directory becomes its owner. The group ID is copied from the parent folder in which the new directory is being created.

The object authority of the owner is set to *OBJMGT + *OBJEXIST + *OBJALTER + *OBJREF.

The primary group and public object authority and all other authorities are copied from the parent folder.

The owner, primary group, and public data authority (including *OBJOPR) are derived from the permissions specified in *mode* (except those permissions that are also set in the file mode creation mask).

The primary group authorities specified in *mode* are not saved if no primary group exists.

5. QOPT File System Differences

When the volume on which the directory is being created is formatted in Universal Disk Format (UDF):

- The authorization that is checked for the object and preceding directories in the path name follows the rules described in [Authorization Required for mkdir\(\)](#).
- The volume authorization list is checked for *CHANGE authority.
- The user who creates the file becomes its owner.
- The group ID is copied from the parent directory in which the file is created.
- The owner, primary group, and public data authorities (*R, *W, and *X) are derived from the permissions specified in the mode.
- The same uppercase and lowercase forms in which the names are entered are preserved. No

distinction is made between uppercase and lowercase when searching for names.

When the volume on which the directory is being created is not formatted in Universal Disk Format (UDF):

- No authorization is checked on the object or preceding directories in the path name.
- The volume authorization list is checked for *CHANGE authority.
- QDFTOWN becomes the owner of the directory.
- No group ID is assigned to the directory.
- The permissions specified in the mode are ignored. The owner, primary group, and public data authorities are set to RWX.
- For newly created directories, names are created in uppercase. No distinction is made between uppercase and lowercase when searching for names.

A directory cannot be created as a direct child of /QOPT.

The change and modification times of the parent directory are not updated.

6. Network File System Differences

Local access to remote directories through the Network File System may produce unexpected results due to conditions at the server. The creation of a directory may fail if permissions and other attributes that are stored locally by the Network File System are more restrictive than those at the server. A later attempt to create a file can succeed when the locally stored data has been refreshed. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) The creation can also succeed after the file system has been remounted.

If you try to re-create a directory that was recently deleted, the request may fail because data that was stored locally by the Network File System still has a record of the directory's existence. The creation succeeds when the locally stored data has been updated.

7. QNetWare File System Differences

The QNetWare file system does not fully support mode bits. See [NetWare on iSeries](#) for more information.

8. QNTC File System Differences

Directory authorities are inherited from the access control list (if any exists) of the parent directory. The mode bits are ignored.

In addition to the normal **mkdir()** function, in the QNTC file system, **mkdir()** can be used to add a server directory under the /QNTC directory level. Directories for all functional Windows NT servers in the local subnet are automatically created. However, Windows NT servers outside the local subnet must be added by using **mkdir()** or the MKDIR command. For example:

```
char new_dir[] = "/QNTC/NTSRV1";  
mkdir(new_dir, NULL)
```

would add the NTSRV1 server into the QNTC directory structure for future access of files and directories on that server.

It is also possible to add the server by using the TCP/IP address. For example:

```
char new_dir[] = "/QNTC/9.130.67.24";
mkdir(new_dir, NULL)
```

The directories added using **mkdir()** will not persist across IPLs. Thus, **mkdir()** or the Make Directory (MKDIR) command must be reissued after every system IPL.

Related Information

- The `<sys/stat.h>` file (see [Header Files for UNIX-Type Functions](#))
- [chmod\(\)--Change File Authorizations](#)
- [QlgMkdir\(\)--Make Directory](#)
- [stat\(\)--Get File Information](#)
- [umask\(\)--Set Authorization Mask for Job](#)
- [pathconf\(\)--Get Configurable Path Name Variables](#)

Example

The following example creates a new directory:

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

main() {
    char new_dir[] = "new_dir";

    if (mkdir(new_dir, S_IRWXU|S_IRGRP|S_IXGRP) != 0)
        perror("mkdir() error");
    else if (chdir(new_dir) != 0)
        perror("first chdir() error");
    else if (chdir("../") != 0)
        perror("second chdir() error");
    else if (rmdir(new_dir) != 0)
        perror("rmdir() error");
    else
        puts("success!");
}
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

mkfifo()--Make FIFO Special File

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);
```

Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **mkfifo()** function creates a new FIFO special file (FIFO) whose name is defined by *path*. A FIFO special file is a type of file with the property that data written to the file is read on a first-in-first-out basis. See the **open()**, **read()**, **write()**, **lseek**, and **close** functions for more characteristics of a FIFO special file.

A FIFO may be opened for reading only or writing only for a uni-directional I/O. It also may be opened for reading and writing access to provide a bi-directional FIFO descriptor.

The file permission bits in *mode* are modified by the file creation mask of the job and then used to set the file permission bits of the FIFO being created.

For more information on the permission bits in *mode*, see [chmod\(\)](#)--Change File Authorizations. For more information on the file creation mask, see [umask\(\)](#)--Set Authorization Mask for Job.

The owner ID of the new FIFO is set to the effective user ID (UID) of the thread. **»**If the object is being created in the Root ('/'), QOpensys, and user-defined file systems, the following applies. If the S_ISGID bit of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the object. If the S_ISGID bit of the parent directory is on, the group ID (GID) of the new object is set to the GID of the parent directory. For all other file systems, the **«**group ID (GID) of the new FIFO is set to the GID of the parent directory.

Upon successful completion, **mkfifo()** sets the access, change, modification, and creation times for the new FIFO. It also sets the change and modification times for the directory that contains the new FIFO (parent directory).

If *path* contains a symbolic link, the symbolic link is followed.

If *path* names a symbolic link, the symbolic link is not followed, and **mkfifo()** fails with the [EEXIST] error.

If bits in *mode* other than the file permission bits are set, **mkfifo()** fails with the [EINVAL] error.

Parameters

path

(Input) A pointer to the null-terminated path name of the FIFO special file to be created.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

The name of the new FIFO is assumed to be represented in the language and country or region currently in effect for the process.

See [QlgMkfifo\(\)](#)--Make FIFO Special File (using NLS-enabled path name) for a description and an example of supplying the *path* in any CCSID.

mode

(Input) Permission bits for the new FIFO.

Authorities

Adopted authority is not used.

Authorization Required for `mkfifo()`

Object Referred to	Authority Required	errno
Each directory in the path name preceding the FIFO to be created.	*X	EACCES
Parent directory of FIFO to be created	*WX	EACCES

Return Value

- 0* `mkfifo()` was successful. The FIFO was created.
- 1* `mkfifo()` was not successful. The FIFO was not created. The *errno* global variable is set to indicate the error.

Error Conditions

If `mkfifo()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file

permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file also may fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EEXIST]

File exists.

The file specified already exists and the specified operation requires that it not exist.

The named file, directory, or path already exists. Or, the last component of *path* is a symbolic link.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than POSIX_SYMLOOP (defined in the limits.h header file). Symbolic links are encountered during resolution of the directory or path name.

[EMLINK]

Maximum link count for a file was exceeded.

An attempt was made to have the link count of a single file exceed LINK_MAX. The value of LINK_MAX can be determined using the pathconf() or the fpathconf() function.

[ENAMETOOLONG]

A path name is too long.

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

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOSYS]

Function not implemented.

An attempt was made to use a function that is not available in this implementation for any object or any arguments.

The path name given refers to an object that does not support this function.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[EPERM]

Operation not permitted.

You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.

[EROFS]

Read-only file system.

You have attempted an update operation in a file system that only supports read operations.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]



The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

- CPE3418 E Possible APAR condition or hardware failure.
- CPFA0D4 E File system error occurred. Error number &1.
- CPF3CF2 E Error(s) occurred during running of &1 API.
- CPF9872 E Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - Independent ASP QSYS.LIB 
 - QOPT

2. File System Differences

The following file systems support **mkfifo()**:

- Root
 - QOpenSys
 - User-defined
3. There are some restrictions when opening a FIFO for text conversion and the CCSIDs involved are not strictly single-byte:
 - Opening a FIFO for reading or reading and writing is not allowed.
 - Any conversion between CCSIDs that are not strictly single-byte must be done by an open instance that has write-only access.
 4. The owner, primary group, and public object authorities (*OBJEXIST, *OBJMGT, *OBJALTER, and *OBJREF) are copied from the parent directory's owner, primary group, and public object authorities. This occurs even when the new FIFO has a different owner than the parent directory. The owner, primary group, and public data authorities (*R, *W, and *X) are derived from the permissions specified in the mode (except for those permissions that are also set in the file mode creation mask). The new FIFO does not have any private authorities or authorization list. It only has authorities for the owner, primary group, and public.

Related Information

- The <sys/stat.h> file (see [Header Files for UNIX-Type Functions](#))
- The <sys/types.h> file (see [Header Files for UNIX-Type Functions](#))

- [chmod\(\)](#)--Change File Authorizations
- [umask\(\)](#)--Set Authorization Mask for Job
- [Mkfifo\(\)](#)--Make FIFO Special File (using NLS-enabled path name)

Example

The following example creates a new FIFO:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

void main() {
    char *mypath = "/newFIFO";

    if (mkfifo(mypath, S_IRWXU|S_IRWXO) != 0)
        perror("mkfifo() error");
    else
        puts("success!");

    return;
}
```

API introduced: V5R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

mmap()--Memory Map a File

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

void *mmap( void *addr,
            size_t len,
            int protection,
            int flags,
            int fildes,
            off_t off);
```

Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Yes

The `mmap()` function establishes a mapping between a process' address space and a stream file.

The address space of the process from the address returned to the caller, for a length of *len*, is mapped onto a stream file starting at offset *off*.

The portion of the stream file being mapped is from starting offset *off* for a length of *len* bytes. The actual address returned from the function is derived from the values of *flags* and the value specified for *address*.

The **mmap()** function causes a reference to be associated with the file represented by *fildes*. This reference is not removed by subsequent close operations. The file remains referenced as long as a mapping exists over the file.

If a mapping already exists for the portion of the processes address space that is to be mapped and the value `MAP_FIXED` was specified for *flags*, then the previous mappings for the affected pages are implicitly unmapped. If one or more files affected by the implicit unmap no longer have active mappings, these files will be unreferenced as a result of **mmap()**.

The use of the **mmap()** function is restricted by the *QSHRMEMCTL* System Value. When this system value is 0, the **mmap()** function may not create a shared mapping having with *PROT_WRITE* capability. Essentially, this prevents the creation of a memory map that could alter the contents of the stream file being mapped. If the *flags* parameter indicated *MAP_SHARED*, the *prot* parameter specifies *PROT_WRITE* and the *QSHRMEMCTL* system value is 0, then the **mmap()** functions will fail and an error number of *EACCES* results.

When the **mmap()** function creates a memory map, the current value of the *QSHRMEMCTL* system value is stored with the mapping. This further restricts attempts to change the protection of the mapping through the use of the **mprotect** function. Changing the system value only affects memory maps created after the system value is changed.

If the size of the file increases after the **mmap()** function completes, then the whole pages beyond the original end of file will not be accessible using the mapping.

If the size of the mapped file is decreased after **mmap()**, attempts to reference beyond the end of the file are undefined and may result in an MCH0601 exception.

Any data written to that portion of the file that is allocated beyond end-of-file may not be preserved. Changes made beyond end of file using mapped access may not be preserved.

The portion of the file beyond end-of-file is assumed to be zero by the traditional file access APIs such as **read()**, **readv()**, **write()**, **writv()**, and **ftruncate()**. The system may clear a partial page, or whole pages allocated beyond end-of-file. This must be taken into account when directly changing a memory mapped file beyond end-of-file. It is not recommended that data be directly changed beyond end-of-file because the extra space allocated varies and unpredictable results may occur.

The **mmap()** function is only supported for *TYPE2 stream files (*STMF) existing in the root (/), QOpenSys, and User-Defined file systems.

➤ Journaling cannot be started while a file is memory mapped. Likewise, a journaled file cannot be memory mapped. The **mmap()** function will fail with EINVAL if the file is journaled. ⚡

The *off* parameter must be zero or a multiple of the system page size. The `_SC_PAGESIZE` or `_SC_PAGE_SIZE` options on the **sysconf()** function may be used to retrieve the system page size.

Parameters

addr

(Input) The starting address of the memory area to be mapped. If the `MAP_FIXED` value is specified with the *flag* parameter, then *address* must be a multiple of the system page size. Use the `_SC_PAGESIZE` or `_SC_PAGE_SIZE` options of the **sysconf()** API to obtain the actual page size in an implementation-independent manner. When the `MAP_FIXED` flag is specified, this address must not be zero.

len

(Input) The length in bytes to map. A length of zero will result in an error of EINVAL.

protection

(Input) The access allowed to this process for this mapping. Specify `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, or a the inclusive-or of `PROT_READ` and `PROT_WRITE`. You cannot specify a protection value more permissive than the mode in which the file was opened.

The `PROT_WRITE` value requires that the file be opened for write and read access.

The following table shows the symbolic constants allowed for the protection parameter.

Symbolic Constant	Decimal Value	Description
<code>PROT_READ</code>	1	Read access is allowed.
<code>PROT_WRITE</code>	2	Write access is allowed. Note that this value assumes <code>PROT_READ</code> also.
<code>PROT_NONE</code>	8	No data access is allowed.
<code>PROT_EXEC</code>	4	This value is allowed, but is equivalent to <code>PROT_READ</code> .

flags

(Input) Further defines the type of mapping desired. There are actually two independent options

controlled through the *flags* parameter.

The first attribute controls whether or not changes made through the mapping will be seen by other processes. The MAP_PRIVATE option will cause a copy on write mapping to be created. A change to the mapping results in a change to a private copy of the affected portion of the file. These changes cannot be seen by other processes. The MAP_SHARED option provides a memory mapping of the file where changes (if allowed by the *protection* parameter) are made to the file. Changes are shared with other processes when MAP_SHARED is specified.

The second control provided by the *flags* parameter in conjunction with the value of the *addr* parameter influences the address range assigned to the mapping. You may request one of the following address selection modes:

1. An exact address range specification. The system will set up the mapping at this location if the address range is valid. Any mapping in the successfully mapping range that existed prior to the mapping operation is implicitly unmapped by this operation.
2. A suggested address range. The system will select a range close to the suggested range.
3. System selected. The system will select an address range. This usually is used to acquire the initial memory map range. Subsequent ranges can be mapped relative to this range.

The MAP_FIXED flag value specifies that the virtual address has been specified through the *addr* parameter. The **mmap()** function will use the value of *addr* as the starting point of the memory map.

When MAP_FIXED is set in the flags parameter, the system is informed that the return value must be equal to the value of *addr*. An invalid value of *addr* when MAP_FIXED is specified will result in a value of MAP_FAILED, which has a value of 0, for the returned value and the the value of *errno* will be set to EINVAL.

When MAP_FIXED is not specified, a value of zero for parameter *addr* indicates that the system may choose the value for the return value. If MAP_FIXED is not specified and a nonzero value is specified for *addr*, the system will take this as a suggestion to find a contiguous address range close to the specified address.

The following table shows the symbolic constants allowed for the flags parameter.

Symbolic Constant	Decimal Value	Description
MAP_SHARED	4	Changes are shared.
MAP_PRIVATE	2	Changes are private.
MAP_FIXED	1	Parameter <i>addr</i> has exact address

fdes

(Input) An open file descriptor.

off

(Input) The offset into the file, in bytes, where the map should begin.

Authorities

No authority checking is performed by the **mmap()** function because this was done by the **open()** functions which assigned the file descriptor, *filides*, used by the **mmap()** function.

The following table shows the open access intent that is required for the various combinations of the mapping sharing mode and mapping intent.

Mapping Sharing Mode	Mapping Intent	Open access intents allowed
MAP_SHARED	PROT_READ	O_RDONLY or O_RDWR
MAP_SHARED	PROT_WRITE	O_RDWR
MAP_SHARED	PROT_NONE	O_RDONLY or O_RDWR
MAP_PRIVATE	PROT_READ	O_RDONLY or O_RDWR
MAP_PRIVATE	PROT_WRITE	O_RDONLY or O_RDWR
MAP_PRIVATE	PROT_NONE	O_RDONLY or O_RDWR

Return Value

Upon successful completion, the **mmap()** function returns the address at which the mapping was placed; otherwise, it returns a value of MAP_FAILED, which has a value of 0, and sets *errno* to indicate the error. The symbol MAP_FAILED is defined in the header <sys/mman.h>.

If successful, function **mmap()** will never return a value of MAP_FAILED.

If **mmap()** fails for reasons other than EBADF, EINVAL, or ENOTSUP, some of the mappings in the address range starting at *addr* and continuing for *len* bytes may have been unmapped and no new mappings are created.

Error Conditions

When the **mmap()** function fails, it returns MAP_FAILED, which has a value of 0, and sets the *errno* as follows.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

The file referenced by *filides* is not open for read, or the file is not opened for write and PROT_WRITE for a shared mapping is being requested. This error also results when the

QSHRMEMCTL system value is 0 and PROT_WRITE is specified.

[EBADFUNC]

Function parameter in the signal function is not set.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file.

The *fildev* parameter does not refer to an open file descriptor.

[EINVAL]

An invalid parameter was found.

A parameter passed to this function is not valid.

The value of the *addr* parameter is invalid. This can occur when MAP_FIXED is specified and the value of the *addr* parameter is not a multiple of the system page size. This may also occur if the value for parameter *addr* is not a valid VOID* pointer or is not within the range allowed.

This error number is also returned if the value of the *flags* parameter does not indicate either MAP_SHARED or MAP_PRIVATE.

➤ This error number is also returned if the specified file is journaled. ⚡

[ENODEV]

No such device.

The *fildev* parameter does not refer to a *TYPE2 stream file (*STMF) in the root, QOpenSys, or user-defined file systems.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

This can occur if the portion of the local process address space reserved for memory mapping has been exceeded.

When MAP_FIXED is specified, it may also occur if the address range specified by the combination of the *addr* and *len* parameters results in a range outside the range reserved for process local storage.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

An unsupported value, or combination of values, was specified on the *protection* parameter.

[ENXIO]

No such device or address.

The portion of the file, as specified by *off* and *len* is not valid for the current size of the file.

[EOVERFLOW]

Object is too large to process.

The object's data size exceeds the limit allowed by this function.

The value of *off* plus *len* exceeds the maximum offset allowed for the file referenced by *files*.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.


Error Messages

The following messages may be sent from this function.

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. The **msync()** function must be used to write changed pages of a shared mapping to disk. If a system crash occurs before the **msync** function is executed, some data may not be preserved.
2. If the application chooses to mix file access methods such as **read()**, **readv()**, **write()**, or **writv()** with **mmap()**, then the application must ensure proper synchronization. While operations such as **read()** and **write()** are relatively atomic because of internal locking, access through the memory map created by **mmap()** does not synchronize with the **read()**, **readv()**, **write()**, and **writv()** functions. Several synchronization functions are available, including the **fcntl()** API, the **DosDetFileLocks()** API, and the mutex functions. Use one of these synchronization methods around access and modifications if atomic access is required. These techniques also will ensure atomic access in a multiprocessor environment.
3. When using **mmap()**, it is necessary to first make a nonspecific mapping request to generate a valid address. This is easily done by specifying a requested address (*addr*) of **0** and not specifying **MAP_FIXED**. Then, using the returned address *pa* as the new requested address (*addr*) and also specifying **MAP_FIXED** for the *flags* parameter. The example below illustrates how this technique can be applied to achieve a contiguous mapping of several files.
4. ➤ The address pointer returned by **mmap()** can only be used with the V4R4M0 or later versions of the following languages:
 - ILE COBOL
 - ILE RPG

- ILE C if the TERASPACE parameter is used when compiling the program. 

Related Information

- [open\(\)](#)--Open File
- [open64\(\)](#)--Open File (Large File Enabled)
- [mmap64\(\)](#)--Memory Map a Stream File (Large File Enabled)
- [munmap\(\)](#)--Remove Memory Mapping
- [mprotect\(\)](#)--Change Access Protection for Memory Mapping
- [msync\(\)](#)--Synchronize Modified Data with Mapped File

Example

The following example creates two files and then produces a contiguous memory mapping of the first data page of each file using two invocations of **mmap()**.

See [Code disclaimer information](#) for information pertaining to code examples.

```
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/mman.h>

main(void) {

    size_t bytesWritten = 0;
    int my_offset = 0;
    char text1[]="Data for file 1.";
    char text2[]="Data for file 2.";
    int fd1,fd2;
    int PageSize;
    void *address;
    void *address2;
    fd1 = open("/tmp/mmaptest1",
              (O_CREAT | O_TRUNC | O_RDWR),
              (S_IRWXU | S_IRWXG | S_IRWXO) );
    if ( fd1 < 0 )
        perror("open() error");
    else {
```

```

bytesWritten = write(fd1, text1, strlen(text1));
if ( bytesWritten != strlen(text1) ) {
    perror("write() error");
    int closeRC = close(fd1);
    return -1;
}

fd2 = open("/tmp/mmaptest2",
          (O_CREAT | O_TRUNC | O_RDWR),
          (S_IRWXU | S_IRWXG | S_IRWXO) );
if (fd2 < 0 )
    perror("open() error");
else {
    bytesWritten = write(fd2, text2, strlen(text2));
    if ( bytesWritten != strlen(text2) )
        perror("write() error");

    PageSize = (int)sysconf(_SC_PAGESIZE);
    if ( PageSize < 0 ) {
        perror("sysconf() error");
    }
    else {

off_t lastoffset = lseek( fd1, PageSize-1, SEEK_SET);
if (lastoffset < 0 ) {
    perror("lseek() error");
}
else {
bytesWritten = write(fd1, " ", 1);    /* grow file 1 to 1 page. */

off_t lastoffset = lseek( fd2, PageSize-1, SEEK_SET);

bytesWritten = write(fd2, " ", 1);    /* grow file 2 to 1 page. */
/*
 * We want to show how to memory map two files with
 * the same memory map. We are going to create a two page
 * memory map over file number 1, even though there is only
 * one page available. Then we will come back and remap
 * the 2nd page of the address range returned from step 1
 * over the first 4096 bytes of file 2.
 */

int len;

my_offset = 0;
len = PageSize;    /* Map one page */
address = mmap(NULL,
              len,
              PROT_READ,
              MAP_SHARED,
              fd1,
              my_offset );
if ( address != MAP_FAILED ) {
    address2 = mmap( ((char*)address)+PageSize,
                  len,
                  PROT_READ,
                  MAP_SHARED | MAP_FIXED, fd2,

```



```

        my_offset );
    if ( address2 != MAP_FAILED ) {
        /* print data from file 1 */
        printf("\n%s",address);
        /* print data from file 2 */
        printf("\n%s",address2);
    } /* address 2 was okay. */
    else {
        perror("mmap() error=");
    } /* mmap for file 2 failed. */
}
else {
    perror("munmap() error=");
}
/*
 * Unmap two pages.
 */
if ( munmap(address, 2*PageSize) < 0 ) {
    perror("munmap() error");
}
else;

}
}
close(fd2);
unlink( "/tmp/mmaptest2");
}
close(fd1);
unlink( "/tmp/mmaptest1");
}
/*
 * Unmap two pages.
 */
if ( munmap(address, 2*PageSize) < 0 ) {
    perror("munmap() error");
}
else;
}
}

```

Output:

```

Data for file 1
Data for file 2

```

API introduced: V5R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

mmap64()--Memory map a Stream File (Large File Enabled)

Syntax

```
#include <sys/mman.h>

void *mmap64( void *addr,
              size_t len,
              int protection,
              int flags,
              int fildes,
              off64_t off);
```

Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Yes

The **mmap64()** function, similar to the **mmap()** function, is used to establish a memory mapping of a file.

For a discussion of the parameters, authorities required, return values, related information, and examples for **mmap()**, see [mmap\(\)--Memory Map a File](#).

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs normally are hidden. To use the **mmap64()** API, you must compile the source with the `_LARGE_FILE_API` macro defined.
2. All of the usage notes for **mmap()** apply to **mmap64()**.

API introduced: V5R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

mprotect()--Change Access Protection for Memory Mapping

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

int mprotect( void *addr,
              size_t len,
              int protection);
```

Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Yes

The **mprotect()** function is used to change the access protection of a memory mapping to that specified by *protection*. All whole pages of the process's address space, that were established by the **mmap()** function, addressed from *addr* continuing for a length of *len* will be affected by the change of access protection. You may specify `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, or the inclusive or of `PROT_READ` and `PROT_WRITE` as values for the *protect* parameter.

Parameters

addr

(Input) The starting address of the memory region for which the access is to be changed.

The *addr* argument must be a multiple of the page size. The **sysconf()** function may be used to determine the system page size.

len

(Input) The length in bytes of the address range.

protection

(Input) The desired access protection. You may specify `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, or the inclusive or of `PROT_READ` AND `PROT_WRITE` as values for the *protection* argument.

No access through the memory mapping will be permitted if `PROT_NONE` is specified.

Storage associated with the mapping cannot be altered unless the `PROT_WRITE` value is specified.

For shared mappings, `PROT_WRITE` requires that the file descriptor used to establish the map had been opened for write access. A shared mapping is a mapping created with the **MAP_SHARED** value of the flag parameter of the **mmap()** function.

Since private mappings do not alter the underlying file, PROT_WRITE may be specified for a mapping that had been created MAP_PRIVATE and had been opened for read access.

The following table shows the symbolic constants allowed for the protection argument.

Symbolic Constant	Decimal Value	Description
PROT_WRITE	2	Write access allowed.
PROT_READ	2	Read access allowed.
PROT_NONE	8	No access allowed.

Authorities

No authorization is required.

Return Value

Upon successful completion, the **mprotect()** function returns 0. Upon failure, -1 is returned and errno is set to the appropriate error number.

Error Conditions

When the **mprotect()** function fails, it returns -1 and sets the errno variable as follows.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

The *protection* argument specifies a protection that violates the access permission the process has to the underlying mapped file.

If the QSHRMEMCTL system value was 0 at the time the mapping was created, then this continues to limit the allowed access until the mapping is destroyed. An attempt to change the protection of a shared mapping to PROT_WRITE when the QSHRMEMCTL system value had been zero at the time of map creation will result in an errno of EACCES.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

The *addr* argument is not a multiple of the page size.

This error number also may indicate that the value of the *len* argument is 0.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

The *addr* argument is out of the allowed range.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

For **mprotect()** this can be caused by an invalid combination of access requests on the *protection* parameter.

Error Messages

The following messages may be sent from this function.

- CPE3418 E Possible APAR condition or hardware failure.
- CPFA0D4 E File system error occurred. Error number &1.
- CPF3CF2 E Error(s) occurred during running of &1 API.
- CPF9872 E Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. ➤ The address pointer that was returned by **mmap()** can only be used with the V4R4M0 or later versions of the following languages:
 - ILE COBOL
 - ILE RPG
 - ILE C if the TERASPACE parameter is used when compiling the program. ⚡

Related Information

- [open\(\)](#)--Open File
- [open64\(\)](#)--Open File (Large File Enabled)
- [creat\(\)](#)--Create or Rewrite File
- [creat64\(\)](#)--Create or Rewrite a File (Large File Enabled)
- [mmap\(\)](#)--Memory Map a Stream File
- [munmap\(\)](#)--Remove Memory Mapping
- [msync\(\)](#)--Synchronize Modified Data with Mapped File

Example

The following example creates a file, produces a memory mapping of the file using **mmap()**, and then changes the protection of the file using **mprotect()**.

```
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/mman.h>

main(void) {

    size_t bytesWritten =0;
    int fd;
    int PageSize;
    char text[] = "This is a test";

    if ( (PageSize = sysconf(_SC_PAGE_SIZE)) < 0) {
        perror("sysconf() Error=");
        return -1;
    }

    fd = open("/tmp/mprotectTest",
              (O_CREAT | O_TRUNC | O_RDWR),
              (S_IRWXU | S_IRWXG | S_IRWXO) );
    if ( fd < 0 ) {
        perror("open() error");
        return fd;
    }
}
```

```

off_t lastoffset = lseek( fd, 0, SEEK_SET);
bytesWritten = write(fd, text, strlen(text));
if (bytesWritten != strlen(text) ) {
    perror("write error. ");
    return -1;
}

lastoffset = lseek( fd, PageSize-1, SEEK_SET);
bytesWritten = write(fd, " ", 1); /* grow file to 1 page. */
if ( bytesWritten != 1 ) {
    perror("write error. ");
    return -1;
}
/* mmap the file. */
void *address;
int len;
off_t my_offset = 0;
len = PageSize; /* Map one page */
address =
    mmap(NULL, len, PROT_NONE, MAP_SHARED, fd, my_offset);

if ( address == MAP_FAILED ) {
    perror("mmap error. ");
    return -1;
}

if ( mprotect( address, len, PROT_WRITE) < 0 ) {
    perror("mprotect failed with error:");
    return -1;
}
else (void) printf("%s",address);

close(fd);
unlink("/tmp/mprotectTest");
}

```

API introduced: V5R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

msync()--Synchronize Modified Data with Mapped File

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

int  msync( void *addr,
            size_t len,
            int  flags );
```

Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Yes

The **msync()** function can be used to write modified data from a shared mapping (created using the **mmap()** function) to non-volatile storage or invalidate privately mapped pages. The data located through mapping address *addr* for a length of *len* are either written to disk, or invalidated, depending on the value of *flags* and the private or shared nature of the mapping.

Parameters

addr

The starting address of the memory region to be synchronized to permanent storage. The specified address must be a multiple of the page size.

len

The number of bytes affected. The length must not be zero. If the length is not a multiple of the page size the system will round this value to the next page boundary.

flags

The desired synchronization.

The following table shows the symbolic constants allowed for the flags parameter.

Symbolic Constant	Decimal Value	Description
MS_ASYNC	1	Perform asynchronous writes.
MS_SYNC	2	Perform synchronous writes.
MS_INVALIDATE	4	Invalidate privately cached data

The MS_SYNC and MS_ASYNC options are mutually exclusive. The MS_SYNC and MS_ASYNC options are ignored if the memory map was created with the MAP_PRIVATE option.

The `MS_INVALIDATE` option is used to discard changes made to a memory map created with the `MAP_PRIVATE` option. The private memory map is synchronized with the current data in the file. Any reference subsequent to the execution of the `msync()` function that invalidates a page will result in a reference to the current value of the file. The first modification of a page after the privately mapped page is invalidated results in the creation of a fresh private copy of that page. Subsequent modifications of this page prior to the next execution of an `msync` that invalidates the page will result in modifications to the same private copy of the page.

The `MS_INVALIDATE` value is ignored if the memory map was created with the `MAP_SHARED` option.

Authorities

No authorization is required.

Return Value

Upon successful completion, the `msync()` function returns 0.

Error Conditions

When the `msync()` function fails, it returns -1 and sets `errno` as follows.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

The value of the *flags* parameter may be invalid.

The value of the *len* parameter may be zero.

The value of the *addr* may not be a multiple of the page size or is out of the allowed range.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

Error Messages

The following messages may be sent from this function.



CPE3418 E Possible APAR condition or hardware failure.
CPFA0D4 E File system error occurred. Error number &1.

CPF3CF2 E Error(s) occurred during running of &1 API.
CPF9872 E Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. The **msync()** function must be used to write changed pages of a shared mapping to disk. If a system crash occurs before the **msync()** function completes, some data may not be preserved.

Process termination does not automatically write changed pages to disk. Some or all pages may be eventually written by the paging subsystem, but no guarantee is given. Therefore, if the data must be preserved the **msync()** function must be used to ensure changes made through a shared memory map are written to disk.

2.  The address pointer that was returned by **mmap()** can only be used with the V4R4M0 or later versions of the following languages:
 - ILE COBOL
 - ILE RPG
 - ILE C if the TERASPACE parameter is used when compiling the program. 

Related Information

- [open\(\)](#)--Open File
- [open64\(\)](#)--Open File (Large File Enabled)
- [mmap\(\)](#)--Memory Map a Stream File
- [munmap\(\)](#)--Remove Memory Mapping
- [mprotect\(\)](#)--Change Access Protection for Memory Mapping

Example

The following example creates a file, creates a memory map, stores data into the file, and writes the data to disk using the **msync()** function.

```
#include <errno.h >
#include <fcntl.h >
#include <unistd.h >
#include <stdio.h >
#include <stdlib.h >
#include <string.h >
#include <sys/types.h >
#include <sys/mman.h >

main(void) {
```

```

size_t bytesWritten =0;
int fd;
int PageSize;
const char textY = "This is a test";

if ( (PageSize = sysconf(_SC_PAGE_SIZE)) < 0 ) {
    perror("sysconf() Error=");
    return -1;
}

fd = open("/tmp/mmsyncTest",
          (O_CREAT | O_TRUNC | O_RDWR),
          (S_IRWXU | S_IRWXG | S_IRWXO) );
if ( fd < 0 ) {
    perror("open() error");
    return fd;
}

off_t lastoffset = lseek( fd, PageSize, SEEK_SET);
bytesWritten = write(fd, " ", 1 );
if (bytesWritten != 1 ) {
    perror("write error. ");
    return -1;
}

    /* mmap the file. */
void *address;
int len;
off_t my_offset = 0;
len = PageSize;    /* Map one page */
address =
    mmap(NULL, len, PROT_WRITE, MAP_SHARED, fd, my_offset);

if ( address == MAP_FAILED ) {
    perror("mmap error. ");
    return -1;
}
    /* Move some data into the file using memory map. */
(void) strcpy( (char*) address, text);
    /* use msync to write changes to disk. */
if ( msync( address, PageSize , MS_SYNC ) < 0 ) {
    perror("msync failed with error:");
    return -1;
}
    else (void) printf("%s","msync completed successfully.");

close(fd);
unlink("/tmp/msyncTest");
}

```

Output:

This is a test.

API introduced: V5R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

munmap()--Remove Memory Mapping

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

int munmap ( void *addr,
             size_t len );
```

Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Yes

The **munmap()** function removes addressability to a range of memory mapped pages of a process's address space. All pages starting with *addr* and continuing for a length of *len* bytes are removed.

The address range specified must begin on a page boundary. Portions of the specified address range which are not mapped, or were not established by the **mmap()** function, are not affected by the **munmap()** function.

If the mapping was created **MAP_PRIVATE** then any private altered pages are discarded and the system storage associated with the copies are returned to the system free space.

When the mapping is removed, the reference associated with the pages mapped over the file is removed. If the file has no references other than those due to memory mapping and the remaining memory mappings are removed by the **munmap()** function, then the file becomes unreferenced. If the file becomes unreferenced due to an **munmap()** function call and the file is no longer linked, then the file will be deleted.

Parameters

addr

The starting address of the memory region being removed.

The *addr* parameter must be a multiple of the page size. The value zero or NULL is not a valid starting address. The **sysconf()** function may be used to determine the system page size.

len

(Input) The length of the address range. All whole pages beginning with *addr* for a length of *len* are included in the address range.

Authorities

No authorization is required.

Return Value

Upon successful completion, the **munmap()** function returns 0. Upon failure, -1 is returned and `errno` is set to the appropriate error number.

Error Conditions

When the **munmap()** function fails, it returns -1 and sets `errno` as follows.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

For **munmap()** this may mean that the address range from *addr* and continuing for a length of *len* is outside the valid range allowed for a process. This error may also indicate that the value for the *addr* parameter is not a multiple of the page size. A value of 0 for parameter *len* also will result in this error number.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

Error Messages

The following messages may be sent from this function.

CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. ➤ The address pointer that was returned by **mmap()** can only be used with the V4R4M0 or later versions of the following languages:
 - ILE COBOL
 - ILE RPG

- ILE C if the TERASPACE parameter is used when compiling the program. <<

Related Information

- [open\(\)](#)--Open File
- [open64\(\)](#)--Open File (Large File Enabled)
- [mmap\(\)](#)--Memory Map a Stream File
- [mprotect\(\)](#)--Change Access Protection for Memory Mapping
- [msync\(\)](#)--Synchronize Modified Data with Mapped File

Example

The following example creates a file, produces a memory mapping of the file using `mmap()`, and then removes the mapping using the `munmap()` function.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/mman.h>

main() {
    char fn[]="creat.file";
    char text[]="This is a test";
    int fd;
    int PageSize;

    if ((fd =
        open(fn, O_CREAT | O_RDWR | O_APPEND, S_IRWXU) < 0)
        perror("open() error");
    else if (write(fd, text, strlen(text)) < 0;
        error("write() error=");
    else if ( (PageSize=sysconf(_SC_PAGESIZE)) < 0 )
        error("sysconf() Error=");
    else {
        off_t lastoffset = lseek( fd, PageSize-1, SEEK_SET);
        write(fd, " ", 1); /* grow file to 1 page. */
        /* mmap the file. */
        void *address;
        int len;
        my_offset = 0;

        len = 4096; /* Map one page */
        address =
            mmap(NULL, len, PROT_READ, MAP_SHARED, fd, my_offset)
        if ( address != MAP_FAILED ) {
```

```
        if ( munmap( address, len ) == -1) {
            error("munmap failed with error:");
        }
    }
    close(fd);
    unlink(fn);
}
}
```

API introduced: V5R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

open()--Open File

Syntax

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

```
int open(const char *path, int oflag, . . .);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see [Usage Notes](#).

The **open()** function opens a file and returns a number called a **file descriptor**. You can use this file descriptor to refer to the file in subsequent I/O operations such as **read()** or **write()**. In these subsequent operations, the file descriptor is commonly identified by the argument *fildev* or *descriptor*. Each file opened by a job gets a new file descriptor.

If the last element of the *path* is a symbolic link, the **open()** function resolves the contents of the symbolic link.

open() positions the **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 the position.

open() clears the FD_CLOEXEC file descriptor flag for the new file descriptor. Refer to [fcntl\(\)--Perform File Control Command](#) for additional information about the FD_CLOEXEC flag.

The **open()** function also can be used to open a directory. The resulting file descriptor can be used in some functions that have a *fildev* parameter.

If the file being opened has been saved and its storage freed, the file is restored during this **open()** function. The storage extension exit program registered against the QIBM_QTA_STOR_EX400 exit point is called to restore the object. (See the [Storage Extension](#) Exit Program for details). If the file cannot successfully be restored, **open()** fails with the EOFFLINE error number.

Parameters

path

(Input) A pointer to the null-terminated path name of the file to be opened.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

When a new file is created, the new file name is assumed to be represented in the language and country or region currently in effect for the job.

See [QlgOpen\(\)--Open a File \(using NLS-enabled path name\)](#) for a description and an example of

supplying the *path* in any CCSID.

oflag

(Input) The file status flags and file access modes of the file to be opened. See [Using the oflag Parameter](#).

Note: The **open64()** API sets the O_LARGEFILE flag internally.

mode

(Input) An optional third parameter of type `mode_t` that is required if the O_CREAT flag is set. It specifies the file permission bits to be used when a file is created. For a description of the permission bits, see [chmod\(\)--Change File Authorizations](#).

conversion ID

(Input) An optional fourth parameter of type unsigned int that is required if the O_CCSDID or O_CODEPAGE flag is set.

If the O_CCSDID flag is set, this parameter specifies a CCSID. If the O_CODEPAGE flag is set, this parameter specifies a code page used to derive a CCSID.

The specified or derived CCSID is assumed to be the CCSID of the data in the file, **»**when a new file is created. This CCSID is associated with the file during file creation.

When the O_TEXT_CREAT flag and its prerequisite flags are not set, the specified or derived CCSID is the CCSID in **«**which data is to be returned (when reading from a file), or the CCSID in which data is being supplied (when writing to a file).

See [Using CCSIDs and code pages](#) for more details.

»*text file creation conversion ID*

(Input) An optional fifth parameter of type unsigned int that is required if the O_TEXT_CREAT flag, along with prerequisite flags O_TEXTDATA, O_CREAT, and either O_CCSDID or O_CODEPAGE, is set. Note: because O_EXCL is not required, this parameter may apply to files that already exist.

When O_CCSDID flag is set, this parameter specifies a CCSID. If the O_CODEPAGE flag is set, this parameter specifies a code page used to derive a CCSID.

The specified or derived CCSID will be used as the CCSID of this open instance. Therefore, this will be the CCSID in which data is to be returned (when reading from a file), or the CCSID in which data is being supplied (when writing to a file). Data will be stored in the CCSID associated with the open file. Note: if the file was not created by this open operation, the file's CCSID may be different than the CCSID associated with the *conversion ID* parameter.

See [Using CCSIDs and code pages](#) for more details. **«**

Using the oflag Parameter

One of the following values *must* be specified in *oflag*:

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 also can be specified in *oflag*:

O_APPEND

Position the file offset at the end of the file before each write operation.

O_CREAT

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 the following description of *O_EXCL*).

If the file being opened does not exist, it is created. The user ID (uid) of the file is set to the effective uid of the job. If the object is being created in the Root ('/'), QOpensys, and user-defined file systems, the following applies. If the *S_ISGID* bit of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the object. If the *S_ISGID* bit of the parent directory is on, the group ID (GID) of the new object is set to the GID of the parent directory. For all other file systems, the group ID (GID) of the file is set to the GID of the directory in which the file is created. File permission bits are set according to *mode*, except for those set in the file mode creation mask of the job. The *S_ISUID* (set-user-ID) and *S_ISGID* (set-group-ID) bits are also set according to *mode*. The file type bits in *mode* are ignored. All other bits in *mode* must be cleared (not set) or a [EINVAL] error is returned.

O_EXCL

Ignored if *O_CREAT* is not set. 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 *path* names a symbolic link, **open()** fails regardless of the contents of the symbolic link.

O_LARGEFILE

Open a large file. The descriptor returned can be used with the other APIs to operate on files larger than 2GB (GB = 1073741824) minus 1 byte. The file systems that do not support large files will just ignore the *O_LARGEFILE* open flag if it is set. The *O_LARGEFILE* flag is ignored by the file systems when **open()** is used to open a directory.

O_TRUNC

Truncate the file to zero length if the file exists and it is a "regular file" (a stream file that can support positioning the file offset). The mode and owner of the file are not changed. *O_TRUNC* applies only to regular files. *O_TRUNC* has no effect on FIFO special files. The *O_TRUNC* behavior applies only when the file is successfully opened with *O_RDWR* or *O_WRONLY*.

Truncation of the file will return the [EOVERFLOW] error if the file is larger than 2 GB minus 1 byte and if the *O_LARGEFILE* oflag is not also specified on the **open()** call. (Note that **open64()** sets the *O_LARGEFILE* oflag automatically.)

If the file exists and it is a regular file, the *S_ISUID* (set-user-ID) and *S_ISGID* (set-group-ID) bits of the file mode are cleared.

If the file has an OS/400 digital signature, **open()** with the **O_TRUNC** oflag causes the signature to be deleted.

O_TEXTDATA

Determines how the data is processed when a file is opened.

- If **O_TEXTDATA** is specified, the data is processed as text.

The data is read from the file and written to the file assuming it is in textual form. When the data is read from the file, it is converted from the **CCSID** of the file to the **CCSID** of the job or the **CCSID** specified by the application receiving the data. When data is written to the file, it is converted to the **CCSID** of the file from the **CCSID** of the job or the **CCSID** specified by the application.

For true stream files, any line-formatting characters (such as carriage return, tab, and end-of-file) are just converted from one **CCSID** to another.

When reading from a record file that is being used as a stream file, end-of-line characters are added to the end of the data in each record. When writing to the record file:

- End-of-line characters are removed.
- Records are padded with blanks (for a source physical file member) or nulls (for a data physical file member).
- Tab characters are replaced by the appropriate number of blanks to the next tab position.

- If **O_TEXTDATA** is not specified, the data is processed as binary. The data is read from the file and written to the file without any conversion. The application is responsible for handling the data.

See [Using CCSIDs and code pages](#) for more details on text conversions.

O_CCSD

The call to open has a fourth argument (*conversion ID*), which is to be interpreted as a **CCSID**. Text conversions between any two **CCSIDs** supported by the **iconv()** API can be performed.

This flag cannot be specified with the **O_CODEPAGE** flag.

See [Using CCSIDs and code pages](#) for more details.

O_CODEPAGE

The call to open has a fourth argument (*conversion ID*), which is to be interpreted as a code page. Only single-byte-to-single-byte or double-byte-to-double-byte text conversions are allowed.

This flag cannot be specified with the **O_CCSD** flag.

See [Using CCSIDs and code pages](#) for more details.

➤ *O_TEXT_CREAT*

The call to open has a fifth argument (*text file creation conversion ID*), which is to be interpreted as either a code page or **CCSID**, depending on whether the **O_CODEPAGE** or **O_CCSD** was set.

If the **O_TEXT_CREAT** flag is specified, all of the following flags must also be specified: **O_CREAT**, **O_TEXTDATA**, and either **O_CODEPAGE** or **O_CCSD**. If all of these prerequisite flags are not specified when **O_TEXT_CREAT** is specified, then the call to open will fail with error condition [EINVAL].

This flag indicates that the textual data read from or written to this file will be converted between the CCSID specified or derived from the *text file creation conversion ID* and the CCSID of the file. When data is read from the file, it is converted from the CCSID of the file to the CCSID specified or derived from the *text file creation conversion ID*. When data is written to the file, it is converted to the CCSID of the file from the CCSID specified or derived from the *text file creation conversion ID*.

See [Using CCSIDs and code pages](#) for more details.❧

O_INHERITMODE

Create the file with the same data authorities as the parent directory that the file is created in. Any data authorities passed in the *mode* parameter are ignored. The mode parameter, however, must still be specified with a valid mode value. This flag is ignored if the O_CREAT flag is not set.

The "root" (/), QOpenSys, QSYS.LIB, ❧independent ASP QSYS.LIB, ❧and QDLS file systems support this flag on an **open()** with the O_CREAT flag set. The QOPT file system ignores this flag because files in this file system do not have data authorities.

O_NONBLOCK

Return without delay from certain operations on this open descriptor.

If O_NONBLOCK is specified when opening a FIFO:

- An **open()** for reading only or reading and writing access returns without delay.
- An **open()** for writing only returns an error if no job currently has the FIFO open for reading. The *errno* value will be ENXIO.

If O_NONBLOCK is not specified when opening a FIFO:

- An **open()** for reading only blocks the calling thread until another thread opens the FIFO for writing.
- An **open()** for writing only blocks the calling thread until another thread opens the FIFO for reading.
- An **open()** for reading and writing returns without delay.

The O_NONBLOCK open flag is ignored for all other object types.

❧*O_SYNC*

Updates to the file will be performed synchronously. All file data and file attributes relative to the I/O operation are written to permanent storage before the update operation returns. Update operations include, but are not limited to, the following: **ftruncate()**, **open()** with O_TRUNC, and **write()**.

O_DSYNC

Updates to the file will be performed synchronously, but only the file data is written to permanent storage before the update operation returns. Update operations include, but are not limited to, the following: **ftruncate()**, **open()** with O_TRUNC, and **write()**.

O_RSYNC

Read operations to the file will be performed synchronously. Pending update requests affecting the data to be read are written to permanent storage. This flag is used in combination with O_SYNC or O_DSYNC. When O_RSYNC and O_SYNC are set, all file data and file attributes are written to permanent storage before the read operation returns. When O_RSYNC and O_DSYNC are set, all file data is written to permanent storage before the read operation returns.❧

A file sharing mode may also be specified in the *oflag*. If none are specified, a default sharing mode of `O_SHARE_RDWR` is used. No more than one of the following may be specified:

O_SHARE_RDONLY

Share with readers only. Open the file only if both of the following are true:

- The file currently is not open for writing.
- The access intent does not conflict with the sharing mode of another open instance of this file.

Once opened with this sharing mode, any request to open this file for writing fails with the [EBUSY] error.

O_SHARE_WRONLY

Share with writers only. Open the file only if both of the following are true:

- The file is not currently open for reading.
- The access intent does not conflict with the sharing mode of another open instance of this file.

Once opened with this sharing mode, any request to open this file for reading fails with the [EBUSY] error.

O_SHARE_RDWR

Share with readers and writers. Open the file only if the access intent of this open does not conflict with the sharing mode of another open instance of this file.

O_SHARE_NONE

Share with neither readers nor writers. Open the file only if the file is not currently open. Once the file is opened with this sharing mode, any request to open this file for reading or writing fails with the [EBUSY] error.

All other bits in *oflag* must be cleared (not set).

Notes:

1. If `O_WRONLY` or `O_RDWR` is specified and the file is checked out by a user profile other than that of the current job, the **open()** fails with the [EBUSY] error.
2. If `O_WRONLY` or `O_RDWR` is specified and the file is marked "read-only," the **open()** fails with the [EROOBJ] error.
3. If `O_CREAT` is specified and the file did not previously exist, a successful **open()** sets the access time, change time, modification time, and creation time for the new file. It also updates the change time and modification time of the directory that contains the new file (the parent directory of the new file).

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

4. Sharing Files

If a sharing mode is not specified in the *oflag* parameter, a default sharing mode of `O_SHARE_RDWR` is used. The **open()** may fail with the [EBUSY] error number if the file is

already open with a sharing mode that conflicts with the access intent of this **open()** request.

Directories may only be opened with a sharing mode of O_SHARE_RDWR. If any other sharing mode is specified, the **open()** fails with error number [EINVAL].

For *CHRSF files, a sharing mode of O_SHARE_RDWR is used regardless of the sharing mode specified in the *oflag* parameter. The sharing mode specified in the *oflag* parameter is ignored.

The following table shows when conflicts will occur:

<i>File Sharing Conflicts</i>				
Access Intent	Sharing Mode			
	Readers Only	Writers Only	Readers and Writers	No Others (Exclusive)
O_RDONLY	OK	EBUSY	OK	EBUSY
O_WRONLY	EBUSY	OK	OK	EBUSY
O_RDWR	EBUSY	EBUSY	OK	EBUSY

Using CCSIDs and code pages

If the O_CCSID or O_CODEPAGE flag is specified, but O_CREAT is not, the *mode* parameter must be specified, but its value will be ignored.

The value of *conversion ID* must be less than 65536. The [EINVAL] error will be returned if it is not.

When a new file is created:

- *conversion ID* is used to derive a CCSID to be associated with the new file (the "file CCSID") and this open instance (the "open CCSID"). If the file is to contain textual data, this CCSID is assumed to be the CCSID of the data, unless the O_TEXT_CREAT flag and its prerequisite flags were also specified.
- If neither O_CCSID nor O_CODEPAGE is specified, or if O_CCSID is specified and *conversion ID* is zero (0), the file CCSID is set to the CCSID of the job. If the job CCSID is 65535, the file CCSID is set to the default CCSID of the job.
- For this open instance, if the O_TEXT_CREAT flag and its prerequisite flags were not specified, the file CCSID and open CCSID are the same and no text conversion will take place on data written to or read from the file, whether O_TEXTDATA is specified or not. If you would like to associate the new file with the CCSID specified in *conversion ID*, but you would also like to have text conversion occur between the file's CCSID and a different CCSID, consider using the O_TEXT_CREAT flag and corresponding *text file creation conversion ID* parameter.
- The QSYS.LIB and independent ASP QSYS.LIB file systems cannot associate the derived CCSID with the database file member being created. Rather, the CCSID of the new member is the CCSID of the database file in which the member is being created. Data read or written during this open instance is converted from or to the CCSID of the database file.

When an existing file is opened:

- *conversion ID* is used to derive a CCSID to be associated with this open instance (the "open CCSID").

- If neither O_CCSID nor O_CODEPAGE is specified, or if O_CCSID is specified and *conversion ID* is zero (0), the open CCSID is set to the CCSID of the job. If the job CCSID is 65535, the open CCSID is set to the default CCSID of the job.
- If O_TEXTDATA is specified, the system will convert from the file CCSID to the open CCSID when reading data from the file, and convert from the open CCSID to the file CCSID when writing data to the file.
- If O_TEXTDATA is specified, but O_CCSID is not:
 - **open()** fails when the file CCSID and open CCSID are not the same and one of them is not strictly single-byte or double-byte.
 - **open()** fails when the file CCSID is double-byte and the open CCSID is single-byte, or the reverse.
 - In either case, the [ECONVERT] error is returned.

See [Examples](#) for a sample program that creates a new file and then opens it for data conversion.

Authorities

Note: Adopted authority is not used.

»Authorization Required for open() (excluding QSYS.LIB, independent ASP QSYS.LIB, and QDLS)«		
Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be opened	*X	EACCES
Existing object when access mode is O_RDONLY	*R	EACCES
Existing object when access mode is O_WRONLY	*W	EACCES
Existing object when access mode is O_RDWR	*RW	EACCES
Existing object when O_TRUNC is specified	*W	EACCES
Parent directory of object to be created when object does not exist and O_CREAT is specified	*WX	EACCES

»Authorization Required for open() in the QSYS.LIB and independent ASP QSYS.LIB File Systems«		
Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be opened	*X	EACCES
Existing object when access mode is O_RDONLY	*R	EACCES
Existing object when access mode is O_WRONLY	*W	EACCES
Existing object when access mode is O_RDWR	*RW	EACCES
Existing object when O_TRUNC is specified	*W	EACCES

Parent directory of object to be created when object does not exist and O_CREAT is specified	*OBJMGT or *OBJALTER	EACCES
Parent directory of the parent directory of object to be created when object does not exist and O_CREAT is specified	*ADD	EACCES

<i>Authorization Required for open() in the QDLS File System</i>		
Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be opened	*X	EACCES
Existing object when access mode is O_RDONLY	*R	EACCES
Existing object when access mode is O_WRONLY	*W	EACCES
Existing object when access mode is O_RDWR	*RW	EACCES
Existing object when O_TRUNC is specified	*W	EACCES
Parent directory of object to be created when object does not exist and O_CREAT is specified	*CHANGE	EACCES

Return Value

value **open()** was successful. The value returned is the file descriptor.

-1 **open()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **open()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

The open sharing mode may conflict with another open of this file, or O_WRONLY or O_RDWR is specified and the file is checked out by another user.

In the QSYS.LIB » and independent ASP QSYS.LIB file systems, « if the O_TEXTDATA flag was specified, the file may be already open in this job or another job where the O_TEXTDATA flag was not specified. Or if the O_TEXTDATA flag was not specified, the file may be already open in this job or another job where the O_TEXTDATA flag was specified.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EEXIST]

File exists.

The file specified already exists and the specified operation requires that it not exist.

The named file, directory, or path already exists.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

- O_RDONLY and O_TRUNC were both specified.
- More than one of O_RDONLY, O_WRONLY, or O_RDWR are set in *oflag*.
- More than one of O_SHARE_RDONLY, O_SHARE_WRONLY, O_SHARE_RDWR, or O_SHARE_NONE are set in *oflag*.
- Unused bits in *oflag* are set and should be cleared.
- Unused bits in *mode* are set and should be cleared.
- It is not valid to open this type of object.
- O_CODEPAGE and O_CCSID were both specified.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[EISDIR]

Specified target is a directory.

The path specified named a directory where a file or object name was expected.

The path name given is a directory. Write access or O_TRUNC has been specified and is not valid for a directory.

➤[EJRNDDAMAGE]

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNENTTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.

[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during

operations that were attempting to send an entry to the journal. ❄

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than `POSIX_SYMLINK` (defined in the `limits.h` header file). Symbolic links are encountered during resolution of the directory or path name.

[EMFILE]

Too many open files for this process.

An attempt was made to open more files than allowed by the value of `OPEN_MAX`. The value of `OPEN_MAX` can be retrieved using the `sysconf()` function.

The process has more than `OPEN_MAX` descriptors already open (see the `sysconf()` function).

[ENAMETOOLONG]

A path name is too long.

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

❄*[ENEWJRN]*

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal. ❄

[ENFILE]

Too many open files in the system.

A system limit has been reached for the number of files that are allowed to be concurrently open in the system.

The entire system has too many other file descriptors already open.

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOSYS]

Function not implemented.

An attempt was made to use a function that is not available in this implementation for any object or any arguments.

The path name given refers to an object that does not support this function.

[ENOSYSRSC]

System resources not available to complete request.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[ENXIO]

No such device or address.

O_NONBLOCK and O_WRONLY open flags are specified, *path* refers to a FIFO, and no job has the FIFO open for reading.

[EOFFLINE]

Operation is suspended.

You have attempted to use an object that has had its data saved and the storage associated with it freed. An attempt to retrieve the object's data failed. The object's data cannot be used until it is restored successfully. The object's data was saved and freed either by saving the object with the STG(*FREE) parameter or by calling an API.

[EOVERFLOW]

Object is too large to process.

The object's data size exceeds the limit allowed by this function.

The size of the specified file cannot be represented correctly in a variable of type `off_t` (the file is larger than 2GB minus 1 byte).

[EPERM]

Operation not permitted.

You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

»[ETXTBSY]

Text file busy.

An attempt was made to execute an OS/400 PASE program that is currently open for writing, or an attempt has been made to open for writing an OS/400 PASE program that is being executed. «

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys

- User-defined
- QNTC
- QSYS.LIB
- »Independent ASP QSYS.LIB «
- QOPT

2. Root, QOpenSys, and User-Defined File System Differences

»The user who creates the file becomes its owner. The `S_ISGID` bit of the directory affects what the group ID (GID) is for objects that are created in the directory. If the `S_ISGID` bit of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the object. If the `S_ISGID` bit of the parent directory is on, the group ID is copied from the parent directory in which the file is created.«

When you do not specify `O_INHERITMODE` for the *oflag* parameter, the owner, primary group, and public object authorities (`*OBJEXIST`, `*OBJMGT`, `*OBJALTER`, and `*OBJREF`) are copied from the parent directory's owner, primary group, and public object authorities. This occurs even when the new file has a different owner than the parent directory. The owner, primary group, and public data authorities (`*R`, `*W`, and `*X`) are derived from the permissions specified in the mode (except for those permissions that are also set in the file mode creation mask). The new file does not have any private authorities or authorization list. It only has authorities for the owner, primary group, and public.

When you specify `O_INHERITMODE` for the *oflag* parameter, the owner, primary group, and public data and object authorities (`*R`, `*W`, `*X`, `*OBJEXIST`, `*OBJMGT`, `*OBJALTER`, and `*OBJREF`) are copied from the parent directory's owner, primary group, and public data and object authorities. In addition, the private authorities (if any) and authorization list (if any) are copied from the parent directory. If the new file has a different owner than the parent directory and the new file's owner has a private authority in the parent directory, that private authority is not copied from the parent directory. The authority for the owner of the new file is copied from the owner of the parent directory.

There are some restrictions when opening a FIFO for text conversion and the CCSIDs involved are not strictly single-byte:

- Opening a FIFO for reading or reading and writing is not allowed. The *errno* global variable is set to `[ENOTSUP]`.
- Any conversion between CCSIDs that are not strictly single-byte must be done by an open instance that has write only access.

3. »QSYS.LIB and Independent ASP QSYS.LIB File System Differences«

The following object types are allowed to be opened:

- `*MBR` (physical file member)

The only types of physical files supported when specifying the `O_TEXTDATA` flag are program-described physical files that contain a single field and source physical files that contain a single text field. Externally described physical files are supported for binary access only.
- `*LIB` (library)
- `*FILE` (physical file or save file)

- *USRSPC (user space)

When a new member is created, the mode and profiles must match those of the parent file. If they do not match, the create operation will fail.

The user who creates a member becomes the owner of the member. However, this owner must be the same as the owner of the parent directory in which the member is being created.

The group ID is obtained from the primary user profile, if one exists. This group ID must be the same as the group ID of the file in which the member is being created.

The primary group authorities specified in *mode* are not saved if no primary group exists.

You cannot open a member in a file that has a mixed data CCSID.

The file access time for a database member is updated using the normal rules that apply to database files. At most, the access time is updated once per day.

Due to the restriction that only one job may have a database member open for writing at a time, the sharing modes O_SHARE_WROONLY and O_SHARE_RDWR do not provide the requested level of sharing.

- If O_SHARE_WROONLY is specified, the **open()** succeeds. However, in all jobs other than the one that performed this **open()**, the actual enforced share mode for this file is equivalent to O_SHARE_NONE.
- If O_SHARE_RDWR is specified, or if no share mode is specified, the **open()** succeeds. However, in all jobs other than the one that performed this **open()**, the actual enforced share mode is equivalent to O_SHARE_RDONLY.

The **open()** of a database member fails with an [EBUSY] error under any of the following conditions:

- The O_TEXTDATA flag is specified, but the file is already open in this job or another job where the O_TEXTDATA flag is not specified.
- The O_TEXTDATA flag is not specified, but the file is already open in this job or another job where the O_TEXTDATA flag and write access are specified.
- The O_TEXTDATA flag is specified and write access is requested, but the file is already open in this job or another job where O_TEXTDATA is specified and write access is also requested.
- ➤ The O_CREAT flag is specified, the member already exists, and the QSYS.LIB or independent ASP QSYS.LIB file system cannot get exclusive access to the member. They must have exclusive access to clear the old member.⚡
- The O_TEXTDATA flag is not specified (binary mode) and more than one job tries to obtain write access to the member. This condition does not apply to PC clients. Because PC clients share the same server job, they can share access to the member.
- The user attempts to open a member with access intentions that conflict with existing object locks on the member.

This function will fail with error code [ENOTSAFE] if the object on which this function is operating is a save file and multiple threads exist in the job.

This function will fail with error code [ENOTSUP] if the file specified is a save file and the O_RDWR flag is specified. A save file can be opened for either reading only or writing only.

This function will fail with error code [ENOTSUP] if the file specified is a save file and the O_TEXTDATA flag is specified.

If a save file containing data is opened for writing, the O_APPEND or O_TRUNC flag must be specified. Otherwise, the **open()** will fail with errno set to [ENOTSUP].

There are some restrictions on sharing modes when opening a save file.

- a. A save file may not have more than one open descriptor per job, regardless of the sharing mode specified.
 - A save file currently open for reading only cannot be opened again in the same job for reading or writing. The **open()** will fail with errno set to [EBUSY].
 - A save file currently open for writing only cannot be opened again in the same job for reading or writing. The **open()** will fail with errno set to [EBUSY].
- b. Due to the restriction that only one job may have a save file open when the save file is open for writing, the sharing modes O_SHARE_WRONLY and O_SHARE_RDWR do not provide the requested level of sharing.
 - If O_SHARE_WRONLY is specified, the **open()** succeeds. However, in all jobs other than the one that performed this **open()**, the actual enforced share mode for this file is equivalent to O_SHARE_NONE.
 - If O_SHARE_RDWR is specified and the file is opened for reading only, the **open()** succeeds. However, in all jobs other than the one that performed this **open()**, the actual enforced share mode is equivalent to O_SHARE_RDONLY.
 - If O_SHARE_RDWR is specified and the file is opened for writing only, the **open()** succeeds. However, in all jobs other than the one that performed this **open()**, the actual enforced share mode is equivalent to O_SHARE_NONE.

Note: Unpredictable results, including loss of data, could occur if, in the same job, a user tries to open the same file for writing at the same time by using both **open()** API for stream file access and a data management open API for record access.

4. QDLS File System Differences

When O_CREAT is specified and a new file is created:



- the owner's object authority is set to *OBJMGT + *OBJEXIST + *OBJALTER + *OBJREF.
- The primary group and public object authority and all other authorities are copied from the directory (folder) in which the file is created.
- The owner, primary group, and public data authority (including *OBJOPR) are derived from the permissions specified in *mode* (except those permissions that are also set in the file mode creation mask).

The primary group authorities specified in *mode* are not saved if no primary group exists.

QDLS does not store the language ID and country or region ID with its files. When this information is requested (using the `readdir()` function), QDLS returns the language ID and country or region ID of the system.

5. QOPT File System Differences

When the volume on which the file is being opened is formatted in Universal Disk Format (UDF):

- The authorization that is checked for the object and preceding directories in the path name follows the rules described in [Authorization Required for open\(\)](#).
- The volume authorization list is checked for `*USE` when the access mode is `O_RDONLY`. The volume authorization list is checked for `*CHANGE` when the access mode is `O_RDWR` or `O_WRONLY`.
- The user who creates the file becomes its owner.
- The group ID is copied from the parent directory in which the file is created.
- The owner, primary group, and public data authorities (`*R`, `*W`, and `*X`) are derived from the permissions specified in the mode (except those permissions that are also set in the file mode creation mask).
- When `O_INHERITMODE` is specified for the `oflag` parameter, the data authorities are copied from the parent directory.
- The sharing modes `O_SHARE_RDONLY`, `O_SHARE_WRONLY`, and `O_SHARE_RDWR` do not provide the requested level of sharing when the access mode is `O_RDWR` or `O_WRONLY`. When the access mode is `O_RDWR` or `O_WRONLY`, the resulting sharing mode semantic will be equivalent to `O_SHARE_NONE`.
- For newly created files, the same uppercase and lowercase forms in which the names are entered are preserved. No distinction is made between uppercase and lowercase when searching for names.
-  This function will fail with error code `[EINVAL]` if the `O_SYNC`, `O_DSYNC`, or `O_RSYNC` open flag is specified. 

When the volume on which the file is being opened is not formatted in Universal Disk Format (UDF):

- No authorization checks are made on the object or preceding directories in the path name.
- The volume authorization list is checked for `*USE` when the access mode is `O_RDONLY`. The volume authorization list is checked for `*CHANGE` when the access mode is `O_RDWR` or `O_WRONLY`.
- `QDFTOWN` becomes the owner of the file.
- No group ID is assigned to the file.

- The permissions specified in the mode are ignored. The owner, primary group, and public data authorities are set to RWX.
- For newly created files, names are created in uppercase. No distinction is made between uppercase and lowercase when searching for names.

6. Network File System Differences

Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. The creation of a file may fail if permissions and other attributes that are stored locally by the Network File System are more restrictive than those at the server. A later attempt to create a file can succeed when the locally stored data has been refreshed. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) The creation can also succeed after the file system has been remounted.

If you try to re-create a file that was recently deleted, the request may fail because data that was stored locally by the Network File System still has a record of the file's existence. The creation succeeds when the locally stored data has been updated.

Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations.

The sharing modes `O_SHARE_RDONLY`, `O_SHARE_WRONLY`, and `O_SHARE_NONE` do not provide the requested level of sharing. If any one of these share modes is specified, the resulting share mode semantic will be equivalent to `O_SHARE_RDWR`.

7. QNetWare File System Differences

The QNetWare file system does not fully support mode bits. See [Netware on iSeries](#) in the iSeries Information Center for more information.

8. This function will fail with the [EOVERFLOW] error if the specified file exists and its size is too large to be represented in a variable of type `off_t` (the file is larger than 2 GB minus 1 byte).
9. When you develop in C-based languages and an application is compiled with the `_LARGE_FILES` macro defined, the `open()` API will be mapped to a call to the `open64()` API.
10. ➤Using this function on the `/dev/null` or `/dev/zero` character special file, the `oflag` values of `O_CREAT` and `O_TRUNC` have no effect. ⚡
11. ➤The `O_SYNC`, `O_DSYNC`, and `O_RSYNC` open flags will not cause updates made to the file by mapped access to be written to permanent storage. ⚡

Related Information

- The `<fcntl.h>` file (see [Header Files for UNIX-Type Functions](#))
- [close\(\)](#)--Close File or Socket Descriptor
- [creat\(\)](#)--Create or Rewrite File
- [dup\(\)](#)--Duplicate Open File Descriptor
- [fcntl\(\)](#)--Perform File Control Command
- [lseek\(\)](#)--Set File Read/Write Offset
- [open64\(\)](#)--Open File (Large File Enabled)
- [OlgOpen\(\)](#)--Open a File (using NLS-enabled path name)
- [read\(\)](#)--Read from Descriptor
- [stat\(\)](#)--Get File Information
- [umask\(\)](#)--Set Authorization Mask for Job
- [write\(\)](#)--Write to Descriptor

Examples


See [Code disclaimer information](#) for information pertaining to code examples.

The following example opens an output file for appending. Because no sharing mode is specified, the `O_SHARE_RDWR` sharing mode is used.

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

The following example creates a new file with read, write, and execute permissions for the user creating the file. If the file already exists, the `open()` fails. If the `open()` succeeds, the file is opened for sharing with readers only.

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

This example first creates an output file for with a specified CCSID. The file is then closed and opened again with data conversion. The `open()` function is called twice because no data conversion would have occurred when using the first open's descriptor on read or write operations, even if `O_TEXTDATA` had been specified on that open; however, the second open could be eliminated entirely by using

O_TEXT_CREAT on the first open. This is demonstrated in the code example immediately following this example. ◀In this example, EBCDIC data is written to the file and converted to ASCII.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int fd;
    int rc;
    char name[]="/test.dat";
    char data[]="abcdefghijk";
    int oflag1 = O_CREAT | O_RDWR | O_CCSID;
    int oflag2 = O_RDWR | O_TEXTDATA | O_CCSID;
    mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR;
    unsigned int file_ccsid = 819;
    unsigned int open_ccsid = 37;

    /******
    /* First create the file with the CCSID 819.          */
    /******

    if ((fd=open(name,oflag1,mode,file_ccsid)) < 0)
    {
        perror("open() for create failed");
        return(0);
    }

    if (close(fd) < 0)
    {
        perror("close() failed.");
        return(0);
    }

    /******
    /* Now open the file so EBCDIC (CCSID 37) data      */
    /* written will be converted to ASCII (CCSID 819).*/
    /******

    if ((fd=open(name,oflag2,mode,open_ccsid)) < 0)
    {
        perror("open() with translation failed");
        return(0);
    }

    /******
    /* Write some EBCDIC data.                          */
    /******

    if (-1 == (rc=write(fd, data, strlen(data))))
    {
        perror("write failed");
        return(0);
    }
}
```

```

if (0 != (rc=close(fd)))
{
    perror("close failed");
    return(0);
}
}

```

➤ In this second example, EBCDIC data is written to the file and converted to ASCII. This will produce the same results as the first example, except that it did it by only using one open instead of two.

```

#include <fcntl.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int fd;
    int rc;
    char name[]="/test.dat";
    char data[]="abcdefghijk";
    int oflag1 = O_CREAT | O_RDWR | O_CCSID | O_TEXTDATA | O_TEXT_CREAT |
O_EXCL;
    mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR;
    unsigned int file_ccsid = 819;
    unsigned int open_ccsid = 37;

    /******
    /* First create the file with the CCSID 819, and */
    /* open it such that the data is converted */
    /* between the the open CCSID of 37 and the */
    /* file's CCSID of 819 when writing data to it. */
    /******

    if ((fd=open(name,oflag1,mode,file_ccsid,open_ccsid)) < 0)
    {
        perror("open() for create failed");
        return(0);
    }

    /******
    /* Write some EBCDIC data. */
    /******

    if (-1 == (rc=write(fd, data, strlen(data))))
    {
        perror("write failed");
        return(0);
    }

    /******
    /* Close the file. */
    /******
    if (0 != (rc=close(fd)))
    {

```

```
    perror("close failed");  
    return(0);  
}  
}  
◀
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

open64()--Open File (Large File Enabled)

Syntax

```
#include <fcntl.h>

int open64(const char *path, int oflag, . . .);
```

Threadsafe: Conditional; see Usage Notes.

The **open64()** function, similar to the **open()** function, opens a file and returns a number called a file descriptor. **open64()** differs from **open()** in that it automatically opens the file with the `O_LARGEFILE` flag set. For a further description of the open flags, see [Using the oflag Parameter](#) in the **open()** API.

For a discussion of the parameters, authorities required, return values, related information, and examples for the **open()** and **open64()** APIs, see [open\(\)--Open File](#).

See [QlgOpen64\(\)--Open File \(Large File Enabled\)](#) for a description and an example of supplying the *path* in any CCSID.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **open64()** API, you must compile the source with the `_LARGE_FILE_API` macro defined.
2. All of the usage notes for **open()** apply to **open64()** and **QlgOpen64()**. See [Usage Notes](#) in the **open()** API.

opendir()--Open Directory

Syntax

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *dirname);

Threadsafe: Conditional; see Usage Notes.
```

The **opendir()** function opens a directory so that it can be read with the **readdir()** function. The variable *dirname* is a string giving the name of the directory to open. If the last component of *dirname* is a symbolic link, **opendir()** follows the symbolic link. As a result, the directory that the symbolic link refers to is opened. The functions **readdir()**, **rewinddir()**, and **closedir()** can be called after a successful call to **opendir()**. The first **readdir()** call reads the first entry in the directory.

Names returned on calls to **readdir()** are returned in the CCSID (coded character set identifier) in effect for the current job at the time this **opendir()** function is called. If the CCSID of the job is 65535, the default CCSID of the job is used. See [QlgOpendir\(\)--Open Directory](#) for specifying a different CCSID.

Parameters

dirname

(Input) A pointer to the null-terminated path name of the directory to be opened.

This parameter is assumed to be represented in the CCSID currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See [QlgOpendir\(\)--Open Directory](#) for a description and an example of supplying the *dirname* in any CCSID.

Authorities

Note: Adopted authority is not used.

Authorization required for opendir()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the directory to be opened	*X	EACCES
The directory to be opened	*R	EACCES

Return Value

value

opendir() was successful. The value returned is a pointer to a DIR, representing an open directory stream. This DIR describes the directory and is used in subsequent operations on the directory using the **readdir()**, **rewinddir()**, and **closedir()** functions.

NULL pointer

opendir() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **opendir()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EEXIST]

File exists.

The file specified already exists and the specified operation requires that it not exist.

The named file, directory, or path already exists.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

 *[EJRNDAMAGE]*

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence

number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNENTTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.

[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal. <<

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than POSIX_SYMLOOP (defined in the limits.h header file). Symbolic links are encountered during resolution of the directory or path name.

[EMFILE]

Too many open files for this process.

An attempt was made to open more files than allowed by the value of OPEN_MAX. The value of OPEN_MAX can be retrieved using the sysconf() function.

The process has more than OPEN_MAX descriptors already open (see the **sysconf()** function).

[ENAMETOOLONG]

A path name is too long.

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

[ENEWJRN]

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal.❏

[ENFILE]

Too many open files in the system.

A system limit has been reached for the number of files that are allowed to be concurrently open in the system.

The entire system has too many other file descriptors already open.

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E Possible APAR condition or hardware failure.
CPFA0D4 E File system error occurred. Error number &1.
CPF3CF2 E Error(s) occurred during running of &1 API.
CPF9872 E Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - >>Independent ASP QSYS.LIB <<
 - QOPT
2. The **opendir()** function uses a file descriptor for each open directory. Applications are limited to opening no more than OPEN_MAX files and directories, and are subject to receiving the [EMFILE] and [ENFILE] errors when too many file descriptors are in use. See the **sysconf()**

function for a description of OPEN_MAX.

The file descriptor that is used by **opendir()** will not be inherited in a child process that is created by the **spawn()** or **spawnp()** API.

3. **opendir()** may allocate memory from the user's heap.
4. Files that are added to the directory after the first call to **readdir()** following an **opendir()** or **rewinddir()** may not be returned on calls to **readdir()**, and files that are removed may still be returned on calls to **readdir()**.
5. QDLS File System Differences

QDLS updates the access time on **opendir()**.

6. QOPT File System Differences

If the directory exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for the directory and preceding directories in the path name follows the rules described in [Authorization required for opendir\(\)](#). If the directory exists on a volume formatted in some other media format, no authorization checks are made on the directory being opened and each directory in the path name. The volume authorization list is checked for *USE authority regardless of the volume media format.

Related Information

- The `<sys/types.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<dirent.h>` file (see [Header Files for UNIX-Type Functions](#))
- [QlgOpendir\(\)--Open Directory](#)
- [readdir\(\)--Read Directory Entry](#)
- [readdir_r\(\)--Read Directory Entry](#)
- [readdir_r_ts64\(\)--Read Directory Entry](#)
- [rewinddir\(\)--Reset Directory Stream to Beginning](#)
- [closedir\(\)--Close Directory](#)
- [spawn\(\)--Spawn Process](#)
- [spawnp\(\)--Spawn Process with Path](#)

Example

The following example opens a directory:

```
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>
```

```

#include <stdio.h>

void traverse(char *fn, int indent) {
    DIR *dir;
    struct dirent *entry;
    int count;
    char path[1025]; /** EXTRA STORAGE MAY BE NEEDED ***/
    struct stat info;

    for (count=0; count<indent; count++) printf("  ");
    printf("%s\n", fn);

    if ((dir = opendir(fn)) == NULL)
        perror("opendir() error");
    else {
        while ((entry = readdir(dir)) != NULL) {
            if (entry->d_name[0] != '.') {
                strcpy(path, fn);
                strcat(path, "/");
                strcat(path, entry->d_name);
                if (stat(path, &info) != 0)
                    fprintf(stderr, "stat() error on %s: %s\n", path,
                            strerror(errno));
                else if (S_ISDIR(info.st_mode))
                    traverse(path, indent+1);
            }
        }
        closedir(dir);
    }
}

main() {
    puts("Directory structure:");
    traverse("/etc", 0);
}

```

Output:

```

Directory structure:
/etc
  /etc/samples
    /etc/samples/IBM
  /etc/IBM

```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

pathconf()--Get Configurable Path Name Variables

Syntax

```
#include <unistd.h>

long pathconf(const char *path, int name);

Threadsafe: Conditional; see Usage Notes.
```

The **pathconf()** function lets an application determine the value of a configuration variable (*name*) associated with a particular file or directory (*path*).

If the named file is a symbolic link, **pathconf()** resolves the symbolic link.

Parameters

path

(Input) A pointer to the null-terminated path name of the file for which the value of the configuration variable is requested.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the process. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See [QlgPathconf\(\)--Get Configurable Path Name Variables](#) for a description and an example of supplying the *path* in any CCSID.

name

(Input) The name of the configuration variable value requested.

The value of *name* can be any one of the following set of symbols defined in the `<unistd.h>` header file, each standing for a configuration variable:

`_PC_LINK_MAX`

Represents `LINK_MAX`, which indicates the maximum number of links the file can have. If *path* is a directory, **pathconf()** returns the maximum number of links that can be established to the directory itself.

`_PC_MAX_CANON`

Represents `MAX_CANON`, which indicates the maximum number of bytes in a terminal canonical input line.

`_PC_MAX_INPUT`

Represents `MAX_INPUT`, which indicates the minimum number of bytes for which space is available in a terminal input queue. This available space is the maximum number of bytes that a portable application can have the user enter before the application actually reads the input.

`_PC_NAME_MAX`

Represents `NAME_MAX`, which indicates the maximum number of bytes in a file name (not including any terminating null at the end if the file name is stored as a string). This symbol refers only to the file name itself; that is, the last component of the path name of the file. `pathconf()` returns the maximum length of file names, even when the path does not refer to a directory.

This value is the number of bytes allowed in the file name if it were encoded in the CCSID of the job. If the CCSID is mixed, this number is an estimate and may be larger than the actual allowable maximum.

`_PC_PATH_MAX`

Represents `PATH_MAX`, which indicates the maximum number of bytes in a complete path name (not including any terminating null at the end if the path name is stored as a string). `pathconf()` returns the maximum length of a relative path name relative to *path*, even when *path* does not refer to a directory.

This value is the number of bytes allowed in the path name if it were encoded in the CCSID of the job. If the CCSID is mixed, this number is an estimate and may be larger than the actual allowable maximum.

`_PC_PIPE_BUF`

Represents `PIPE_BUF`, which indicates the maximum number of bytes that can be written "atomically" to a pipe. If more than this number of bytes are written to a pipe, the operation may take more than one physical write operation and physical read operation to read the data on the other end of the pipe. If *path* is a FIFO special file, `pathconf()` returns the value for the file itself. If *path* is a directory, `pathconf()` returns the value for any FIFOs that exist or that can be created under the directory. If *path* is any other kind of file, an error of `[EINVAL]` is returned.

`_PC_CHOWN_RESTRICTED`

Represents `_POSIX_CHOWN_RESTRICTED`, as defined in the `<unistd.h>` header file. It restricts use of `chown()` to a job with appropriate privileges, and allows the group ID of a file to be changed only to the effective group ID of the job or to one of its supplementary group IDs. If *path* is a directory, `pathconf()` returns the value for any kind of file under the directory, but not for subdirectories of the directory.

`_PC_NO_TRUNC`

Represents `_POSIX_NO_TRUNC`, as defined in the `<unistd.h>` header file. It generates an error if a file name is longer than `NAME_MAX`. If *path* refers to a directory, the value returned by `pathconf()` applies to all files under that directory.

`_PC_VDISABLE`

Represents `_POSIX_VDISABLE`, as defined in the `<unistd.h>` header file. This symbol indicates that terminal special characters can be disabled using this character value, if it is defined.

`_PC_THREAD_SAFE`

This symbol is used to determine if the object represented by *path* resides in a threadsafe file system. `pathconf()` returns the value 1 if the file system is threadsafe and 0 if the file system is not threadsafe. `fpathconf()` will never fail with error code `[ENOTSAFE]` when called with `_PC_THREAD_SAFE`.

Authorities

Note: Adopted authority is not used.

Authorization required for `pathconf()`

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object	None	None

Return Value

value

pathconf() was successful. The value of the variable requested in *name* is returned.

-1

One of the following has occurred:

- A particular variable has no limit (for example, `_PC_PATH_MAX`). The *errno* global variable is not changed.
- **pathconf()** was not successful. The *errno* is set.

Error Conditions

If **fpathconf()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBADNAME]

The object name specified is not correct.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[ECONVERT]

Conversion error.

One or more characters could not be converted from the source CCSID to the target CCSID.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFILECVT]

File ID conversion of a directory failed.

Try to run the Reclaim Storage (RCLSTG) command to recover from this error.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL. *name* is not a valid configuration variable name, or the given variable cannot be associated with the specified file.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[EISDIR]

Specified target is a directory.

The path specified named a directory where a file or object name was expected.

The path name given is a directory.

[ELOOP]

A loop exists in the symbolic links.

This error is issued if the number of symbolic links encountered is more than `POSIX_SYMLoop` (defined in the `limits.h` header file). Symbolic links are encountered during resolution of the directory or path name.

[ENAMETOOLONG]

A path name is too long.

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

[ENOENT]

No such path or directory.

The directory or a component of the path name specified does not exist.

A named file or directory does not exist or is an empty string.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTDIR]

Not a directory.

A component of the specified path name existed, but it was not a directory when a directory was expected.

Some component of the path name is not a directory, or is an empty string.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENOTSUP]

Operation not supported.

The operation, though supported in general, is not supported for the requested object or the requested arguments.

[EPERM]

Operation not permitted.

You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.

[EROOBJ]

Object is read only.

You have attempted to update an object that can be read only.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. When this function is called with any configuration variable name except `_PC_THREAD_SAFE`, the following usage note applies:
 - This function will fail with error code `[ENOTSAFE]` when all the following conditions are

true:

- Where multiple threads exist in the job.
- The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - »Independent ASP QSYS.LIB«
 - QOPT

Related Information

- The <**unistd.h**> file (see [Header Files for UNIX-Type Functions](#))
- [chown\(\)--Change Owner and Group of File](#)
- [fpathconf\(\)--Get Configurable Path Name Variables by Descriptor](#)
- [QlgPathconf\(\)--Get Configurable Path Name Variables](#)

Example

The following example determines the maximum number of bytes in a file name:

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

main() {
    long result;

    errno = 0;
    puts("examining NAME_MAX limit for root filesystem");
    if ((result = pathconf("/", _PC_NAME_MAX)) == -1)
        if (errno == 0)
            puts("There is no limit to NAME_MAX.");
        else perror("pathconf() error");
    else
        printf("NAME_MAX is %ld\n", result);
}
```

Output:

```
examining NAME_MAX limit for root filesystem
NAME_MAX is 255
```

API introduced: V3R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

pipe()--Create an Interprocess Channel

Syntax

```
#include <unistd.h>

int pipe(int fildes[2]);
Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Yes
```

The **pipe()** function creates a data pipe and places two file descriptors, one each into the arguments *fildes[0]* and *fildes[1]*, that refer to the open file descriptions for the read and write ends of the pipe, respectively. Their integer values will be the two lowest available at the time of the **pipe()** call. The `O_NONBLOCK` and `FD_CLOEXEC` flags will be clear on both descriptors. NOTE: these flags can, however, be set by the **fcntl()** function.

Data can be written to the file descriptor *fildes[1]* and read from file descriptor *fildes[0]*. A read on the file descriptor *fildes[0]* will access data written to the file descriptor *fildes[1]* on a first-in-first-out basis. File descriptor *fildes[0]* is open for reading only. File descriptor *fildes[1]* is open for writing only.

The **pipe()** function is often used with the **spawn()** function to allow the parent and child processes to send data to each other.

Upon successful completion, **pipe()** will update the access time, change time, and modification time of the pipe.

Parameters

fildes[2]

(Output) An integer array of size 2 that will receive the pipe descriptors.

Authorities

None.

Return Value

0 **pipe()** was successful.

-1 **pipe()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If `pipe()` is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

- [EFAULT]* The address used for an argument is not correct.
- In attempting to use an argument in a call, the system detected an address that is not valid.
- While attempting to access a parameter passed to this function, the system detected an address that is not valid.
- [EMFILE]* Too many open files for this process.
- An attempt was made to open more files than allowed by the value of `OPEN_MAX`. The value of `OPEN_MAX` can be retrieved using the `sysconf()` function.
- The process has more than `OPEN_MAX` descriptors already open (see the `sysconf()` function).
- [ENFILE]* Too many open files in the system.
- A system limit has been reached for the number of files that are allowed to be concurrently open in the system.
- The entire system has too many other file descriptors already open.
- [ENOMEM]* Storage allocation request failed.
- A function needed to allocate storage, but no storage is available.
- There is not enough memory to perform the requested function.
- [EUNKNOWN]* Unknown system state.
- The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

Related Information

- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<fcntl.h>` file (see [Header Files for UNIX-Type Functions](#))
- [fcntl\(\)--Perform File Control Command](#)
- [fstat\(\)--Get File Information by Descriptor](#)
- [Qp0zPipe\(\)--Create Interprocess Channel with Sockets](#)
- [read\(\)--Read from Descriptor](#)
- [spawn\(\)--Spawn Process](#)
- [write\(\)--Write to Descriptor](#)

Example

See [Code disclaimer information](#) for information pertaining to code examples.

The following example creates a pipe, writes 10 bytes of data to the pipe, and then reads those 10 bytes of data from the pipe.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

void main()
{
    int fildes[2];
    int rc;
    char writeData[10];
    char readData[10];
    int bytesWritten;
    int bytesRead;

    memset(writeData, 'A', 10);

    if (-1 == pipe(fildes))
    {
        perror("pipe error");
        return;
    }

    if (-1 == (bytesWritten = write(fildes[1],
                                   writeData,
                                   10)))
    {
        perror("write error");
    }
    else
    {
        printf("wrote %d bytes\n", bytesWritten);

        if (-1 == (bytesRead = read(fildes[0],
                                   readData,
                                   10)))
        {
            perror("read error");
        }
        else
        {
            printf("read %d bytes\n", bytesRead);
        }
    }

    close(fildes[0]);
    close(fildes[1]);
}
```

```
    return;  
}
```

API introduced: V5R1

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

»pread()--Read from Descriptor with Offset

Syntax

```
#include <unistd.h>

ssize_t pread(int file_descriptor,
              void *buf, size_t nbyte, off_t offset);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

From the file indicated by *file_descriptor*, the **pread()** function reads *nbyte* bytes of input into the memory area indicated by *buf*. The *offset* value defines the starting position in the file and the file pointer position is not changed.

See [read\(\)--Read from Descriptor](#) for more information relating to reading from a descriptor.

In the QSYS.LIB and independent ASP QSYS.LIB file systems, the offset will be ignored for a member while in text mode.

Parameters

file_descriptor

(Input) The descriptor to be read.

buf

(Output) A pointer to a buffer in which the bytes read are placed.

nbyte

(Input) The number of bytes to be read.

offset

(Input) The offset to the desired starting position in the file.

Authorities

No authorization is required.

Return Value

- value* **pread()** was successful. The value returned is the number of bytes actually read and placed in *buf*. This number is less than or equal to *nbyte*. It is less than *nbyte* only if **pread()** reached the end of the file before reading the requested number of bytes. If **pread()** is reading a regular file and encounters a part of the file that has not been written (but before the end of the file), **pread()** places bytes containing zeros into *buf* in place of the unwritten bytes.
- 1 **pread()** was not successful. The *errno* global variable is set to indicate the error. If the value of *nbyte* is greater than `SSIZE_MAX`, **pread()** sets *errno* to `[EINVAL]`.

Error Conditions

If **pread()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADF]

Descriptor not valid.

A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file. Or, this **pread** request was made to a file that was only open for writing.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

This may occur if the file resides in a file system that does not support large files, and the starting offset of the file exceeds 2GB minus 2 bytes.

This will also occur if the *offset* value is less than 0.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENXIO]

No such device or address.

[Eoverflow]

Object is too large to process.

The object's data size exceeds the limit allowed by this function.

The file is a regular file, *nbyte* is greater than 0, the starting offset is before the end-of-file, and the starting offset is greater than or equal to 2GB minus 2 bytes.

[ERestart]

A system call was interrupted and may be restarted.

[ESPIPE]

Seek request not supported for object.

A seek request was specified for an object that does not support seeking.

The object is not capable of seeking.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

- Root
- QOpenSys
- User-defined
- QNTC
- QSYS.LIB
- Independent ASP QSYS.LIB
- QOPT

2. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

This function will fail with error code [ENOTSAFE] if the object on which this function is operation is a save file and multiple threads exist in the job.

This function will fail with error code [EIO] if the file specified is a save file and the file does not contain complete save file data.

The file access time for a database member is updated using the normal rules that apply to database files. At most, the access time is updated once per day.

If you previously used the integrated file system interface to manipulate a member that contains an end-of-file character, you should avoid using other interfaces (such as the Source Entry Utility or database reads and writes) to manipulate the member. If you use other interfaces after using the integrated file system interface, the end-of-file information will be lost.

3. QOPT File System Differences

The file access time is not updated on a **pread()** operation.

When reading from files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being read are ignored.

4. Network File System Differences

Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)

Reading and writing to files with the Network File System relies on byte-range locking to guarantee data integrity. To prevent data inconsistency, use the **fcntl()** API to get and release these locks.

5. QFileSvr.400 File System Differences

The largest buffer size allowed is 16 megabytes. If a larger buffer is passed, the error EINVAL will be received.

6. For file systems that do not support large files, **pread()** will return [EINVAL] if the starting offset exceeds 2GB minus 2 bytes, regardless of how the file was opened. For the file systems that do support large files, **pread()** will return [EOVERFLOW] if the starting offset exceeds 2GB minus 2 bytes and the file was not opened for large file access.

7. Using this function successfully on the `/dev/null` or `/dev/zero` character special file results in a return value of zero. In addition, the access time for the file is updated.
8. If *file_descriptor* refers to a descriptor obtained using the `open()` function with `O_TEXTDATA` and `O_CCSID` specified, the file `CCSID` and open `CCSID` are not the same, and the converted data could expand or contract, then the *offset* value must be 0.
9. If *file_descriptor* refers to a character special file, the *offset* value is ignored.

Related Information

- The `<limits.h>` file (see [Header Files for UNIX-Type Functions](#))
- The `<unistd.h>` file (see [Header Files for UNIX-Type Functions](#))
- [creat\(\)--Create or Rewrite File](#)
- [dup\(\)--Duplicate Open File Descriptor](#)
- [dup2\(\)--Duplicate Open File Descriptor to Another Descriptor](#)
- [fcntl\(\)--Perform File Control Command](#)
- [ioctl\(\)--Perform I/O Control Request](#)
- [lseek\(\)--Set File Read/Write Offset](#)
- [open\(\)--Open File](#)
- [pread64\(\)--Read from Descriptor with Offset \(large file enabled\)](#)
- [pwrite\(\)--Write to Descriptor with Offset](#)
- [pwrite64\(\)--Write to Descriptor with Offset \(large file enabled\)](#)
- [read\(\)--Read from Descriptor](#)
- [readv\(\)--Read from Descriptor Using Multiple Buffers](#)
- [recv\(\)--Receive Data](#)
- [recvfrom\(\)--Receive Data](#)
- [recvmsg\(\)--Receive Data or Descriptors or Both](#)
- [write\(\)--Write to Descriptor](#)
- [writev\(\)--Write to Descriptor Using Multiple Buffers](#)

Example

The following example opens a file and reads input:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
```

```

main() {
    int ret, file_descriptor;
    off_t off=5;
    char buf[]="Test text";

    if ((file_descriptor = creat("test.output", S_IWUSR))!= 0)
        perror("creat() error");
    else {
        if (-1==(rc=write(file_descriptor, buf, sizeof(buf)-1)))
            perror("write() error");
        if (close(file_descriptor)!= 0)
            perror("close() error");
    }

    if ((file_descriptor = open("test.output", O_RDONLY)) < 0)
        perror("open() error");
    else {
        ret = pread(file_descriptor, buf, ((sizeof(buf)-1)-off), off);
        buf[ret] = 0x00;
        printf("block pread: \n<%s>\n", buf);
        if (close(file_descriptor)!= 0)
            perror("close() error");
    }
    if (unlink("test.output")!= 0)
        perror("unlink() error");
}

```

Output:

```

block pread:
<text>

```



API introduced: V5R2

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)



pread64()--Read from Descriptor with Offset (large file enabled)

Syntax

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

```
ssize_t pread64(int file_descriptor,  
                void *buf, size_t nbyte, off64_t offset);
```

Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

From the file indicated by *file_descriptor*, the **pread64()** function reads *nbyte* bytes of input into the memory area indicated by *buf*. The *offset* value defines the starting position in the file and the file pointer position is not changed.

pread64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 byte as long as the file has been opened by either of the following:

- Using the **open64()** function (see [open64\(\)--Open File \(large file enabled\)](#)).
- Using the **open()** function (see [open\(\)--Open File](#)) with `O_LARGEFILE` set in the `oflag` parameter.

For additional information about parameters, authorities, and error conditions, see [pread\(\)--Read from Descriptor with Offset](#).

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **pread64** API, you must compile the source with the `_LARGE_FILE_API` macro defined.
2. All of the usage notes for **pread()** apply to **pread64()**. See *Usage Notes* in the **pread** API.

Example

The following example opens a file and reads input:

```
#define _LARGE_FILE_API  
#include <stdio.h>  
#include <unistd.h>  
#include <fcntl.h>
```



```

main() {
    int ret, file_descriptor;
    off64_t off=5;
    char buf[]="Test text";

    if ((file_descriptor = creat64("test.output", S_IWUSR))!= 0)
        perror("creat64() error");
    else {
        if (-1==(rc=write(file_descriptor, buf, sizeof(buf)-1)))
            perror("write() error");
        if (close(file_descriptor)!= 0)
            perror("close() error");
    }

    if ((file_descriptor = open64("test.output", O_RDONLY)) < 0)
        perror("open64() error");
    else {
        ret = pread64(file_descriptor, buf, ((sizeof(buf)-1)-off), off);
        buf[ret] = 0x00;
        printf("block pread64: \n<%s>\n", buf);
        if (close(file_descriptor)!= 0)
            perror("close() error");
    }
    if (unlink("test.output")!= 0)
        perror("unlink() error");
}

```

Output:

```

block pread64:
<text>

```



API introduced: V5R2

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)



pwrite()--Write to Descriptor with Offset

Syntax

```
#include <unistd.h>

ssize_t pwrite
(int file_descriptor, const void *buf,
 size_t nbyte, off_t offset);
```

Service Program Name: QPOLLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **pwrite()** function writes *nbyte* bytes from *buf* to the file associated with *file_descriptor*. The *offset* value defines the starting position in the file and the file pointer position is not changed.

See [write\(\)--Write to Descriptor](#) for more information relating to writing to a descriptor.

In the QSYS.LIB and independent ASP QSYS.LIB file systems, the *offset* will be ignored for a member while in text mode.

The *offset* will also be ignored if *file_descriptor* refers to a descriptor obtained using the **open()** function with O_APPEND specified.

Parameters

file_descriptor

(Input) The descriptor of the file to which the data is to be written.

buf

(Input) A pointer to a buffer containing the data to be written.

nbyte

(Input) The size in bytes of the data to be written.

offset

(Input) The offset to the desired starting position in the file.

Authorities

No authorization is required.

Return Value

value **pwrite()** was successful. The value returned is the number of bytes actually written. This number is less than or equal to *nbyte*.

-1 **pwrite()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **pwrite()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EACCES]

Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions.

The thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EAGAIN]

Operation would have caused the process to be suspended.

[EBADF]

Descriptor not valid.

A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.

A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file. Or this **pwrite()** request was made to a file that was only open for reading.

[EBADFID]

A file ID could not be assigned when linking an object to a directory.

The file ID table is missing or damaged.

To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.

[EBUSY]

Resource busy.

An attempt was made to use a system resource that is not available at this time.

[EDAMAGE]

A damaged object was encountered.

A referenced object is damaged. The object cannot be used.

[EFAULT]

The address used for an argument is not correct.

In attempting to use an argument in a call, the system detected an address that is not valid.

While attempting to access a parameter passed to this function, the system detected an address that is not valid.

[EFBIG]

Object is too large.

The size of the object would exceed the system allowed maximum size or the process soft file size limit.

The file is a regular file, *nbyte* is greater than 0, and the starting offset is greater than or equal to 2 GB minus 2 bytes.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

The file system that the file resides in does not support large files, and the starting offset exceeds 2GB minus 2 bytes.

This will also occur if the *offset* value is less than 0.

[EIO]

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

[EJRNDAMAGE]

Journal damaged.

A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNTTOOLONG]

Entry too large to send.

The journal entry generated by this operation is too large to send to the journal.

[EJRNINACTIVE]

Journal inactive.

The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

[EJRNRCVSPC]

Journal space or system storage error.

The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal.

[ENEWJRN]

New journal is needed.

The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.

[ENEWJRNRCV]

New journal receiver is needed.

A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal.

[ENOMEM]

Storage allocation request failed.

A function needed to allocate storage, but no storage is available.

There is not enough memory to perform the requested function.

[ENOSPC]

No space available.

The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.

Insufficient space remains to hold the intended file, directory, or link.

[ENOTAVAIL]

Independent Auxiliary Storage Pool (ASP) is not available.

The independent ASP is in Vary Configuration (VRYCFG), or Reclaim Storage (RCLSTG) processing.

To recover from this error, wait until processing has completed for the independent ASP.

[ENOTSAFE]

Function is not allowed in a job that is running with multiple threads.

[ENXIO]

No such device or address.

[ERESTART]

A system call was interrupted and may be restarted.

[ETRUNC]

Data was truncated on an input, output, or update operation.

[ESPIPE]

Seek request not supported for object.

A seek request was specified for an object that does not support seeking.

The object is not capable of seeking.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[EUNKNOWN]

Unknown system state.

The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

[EADDRNOTAVAIL]

Address not available.

[ECONNABORTED]

Connection ended abnormally.

[ECONNREFUSED]

The destination socket refused an attempted connect operation.

[ECONNRESET]

A connection with a remote socket was reset by that socket.

[EHOSTDOWN]

A remote host is not available.

[EHOSTUNREACH]

A route to the remote host is not available.

[ENETDOWN]

The network is not currently available.

[ENETRESET]

A socket is connected to a host that is no longer available.

[ENETUNREACH]

Cannot reach the destination network.

[ESTALE]

File or object handle rejected by server.

If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
 - Where multiple threads exist in the job.
 - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
 - Root
 - QOpenSys
 - User-defined
 - QNTC
 - QSYS.LIB
 - Independent ASP QSYS.LIB
 - QOPT

2. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

This function will fail with error code [ENOTSAFE] if the object on which this function is operating is a save file and multiple threads exist in the job.

If the file specified is a save file, only complete records will be written into the save file. A **pwrite()** request that does not provide enough data to completely fill a save file record will cause the partial record's data to be saved by the file system. The saved partial record will then be combined with additional data on subsequent **pwrite()**'s until a complete record may be written into the save file. If the save file is closed prior to a saved partial record being written into the save file, then the saved partial record is discarded, and the data in that partial record will need to be written again by the application.

A successful **pwrite()** updates the change, modification, and access times for a database member using the normal rules that apply to database files. At most, the access time is updated once per day.

You should be careful when writing end-of-file characters in the QSYS.LIB and independent ASP QSYS.LIB file systems. For these file systems, end-of-file characters are symbolic; that is, they are stored outside the file member. However, some situations can result in actual, nonsymbolic end-of-file characters being written to a member. These nonsymbolic end-of-file characters could cause some tools or utilities to fail. For example:

- If you previously wrote an end-of-file character as the last character of a member, do not continue to write data after that end-of-file character. Continuing to write data will cause a nonsymbolic end-of-file to be written. As a result, a compile of the member could fail.
- If you previously wrote an end-of-file character as the last character of a member, do not write other end-of-file characters preceding it in the file. This will cause a nonsymbolic end-of-file to be written. As a result, a compile of the member could fail.
- If you previously used the integrated file system interface to manipulate a member that contains an end-of-file character, avoid using other interfaces (such as the Source Entry Utility or database reads and writes) to manipulate the member. If you use other interfaces after using the integrated file system interface, the end-of-file information will be lost.

3. QOPT File System Differences

The change and modification times of the file are updated when the file is closed.

When writing to files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being written are ignored.

4. Network File System Differences

Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations (several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data).

Reading and writing to files with the Network File System relies on byte-range locking to guarantee data integrity. To prevent data inconsistency, use the **fcntl()** API to get and release these locks.

5. QFileSvr.400 File System Differences

The largest buffer size allowed is 16 megabytes. If a larger buffer is passed, the error EINVAL will be received.

6. For the file systems that do not support large files, **pwrite()** will return [EINVAL] if the starting offset exceeds 2GB minus 2 bytes, regardless of how the file was opened. For the file systems that do support large files, **pwrite()** will return [EFBIG] if the starting offset exceeds 2GB minus 2 bytes and the file was not opened for large file access.
7. Using this function successfully on the /dev/null or /dev/zero character special file results in a return value of the total number of bytes requested to be written. No data is written to the /dev/null or /dev/zero character special file. In addition, the change and modification times for the file are updated.
8. If the write exceeds the process soft file size limit, signal SIFXFSZ is issued.
9. If *file_descriptor* refers to a descriptor obtained using the **open()** function with O_TEXTDATA and O_CCSID specified, the file CCSID and open CCSID are not the same, and the converted data could expand or contract, then the *offset* value must be 0.
10. If *file_descriptor* refers to a character special file, the *offset* value is ignored.

Related Information

- The <fcntl.h> file (see [Header Files for UNIX-Type Functions](#))
- The <unistd.h> file (see [Header Files for UNIX-Type Functions](#))
- [creat\(\)](#)--Create or Rewrite File
- [dup\(\)](#)--Duplicate Open File Descriptor
- [dup2\(\)](#)--Duplicate Open File Descriptor to Another Descriptor
- [fcntl\(\)](#)--Perform File Control Command
- [ioctl\(\)](#)--Perform I/O Control Request
- [lseek\(\)](#)--Set File Read/Write Offset
- [open\(\)](#)--Open File
- [pread\(\)](#)--Read from Descriptor with Offset
- [pread64\(\)](#)--Read from Descriptor with Offset (large file enabled)
- [pwrite64\(\)](#)--Write to Descriptor with Offset (large file enabled)
- [read\(\)](#)--Read from Descriptor
- [readv\(\)](#)--Read from Descriptor Using Multiple Buffers
- [send\(\)](#)--Send Data
- [sendmsg\(\)](#)--Send Data or Descriptors or Both
- [sendto\(\)](#)--Send Data
- [write\(\)](#)--Write to Descriptor
- [writev\(\)](#)--Write to Descriptor Using Multiple Buffers

Example

The following example writes a specific number of bytes to a file:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

#define mega_string_len 1000000

main() {
    char *mega_string;
    int file_descriptor;
    int ret;
    off_t off=5;
    char fn[]="write.file";

    if ((mega_string = (char*) malloc(mega_string_len+off)) == NULL)
        perror("malloc() error");
    else if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        memset(mega_string, '0', mega_string_len);
        if ((ret = pwrite(file_descriptor, mega_string, mega_string_len, off))
== -1)
            perror("pwrite() error");
        else printf("pwrite() wrote %d bytes at offset %d\n", ret, off);
        if (close(file_descriptor)!= 0)
            perror("close() error");
        if (unlink(fn)!= 0)
            perror("unlink() error");
    }
}
```

Output:

```
pwrite() wrote 1000000 bytes at offset 5
```



API introduced: V5R2

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)



pwrite64()--Write to Descriptor with Offset (large file enabled)

Syntax

```
#include <unistd.h>

ssize_t pwrite64
    (int file_descriptor, const void *buf,
     size_t nbyte, off64_t offset);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **pwrite64()** function writes *nbyte* bytes from *buf* to the file associated with *file_descriptor*. The *offset* value defines the starting position in the file and the file pointer position is not changed.

In the QSYS.LIB and independent ASP QSYS.LIB file systems, the *offset* will be ignored for a member while in text mode.

The *offset* will also be ignored if *file_descriptor* refers to a descriptor obtained using the **open()** function with `O_APPEND` specified.

pwrite64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 byte as long as the file has been opened by either of the following:

- Using the **open64()** function (see [open64\(\)--Open File \(large file enabled\)](#)).
- Using the **open()** function (see [open\(\)--Open File](#)) with `O_LARGEFILE` set in the *oflag* parameter.

For additional information about parameters, authorities, and error conditions, see [pwrite\(\)--Write to Descriptor with Offset](#).

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **pwrite64** API, you must compile the source with the `_LARGE_FILE_API` macro defined.
2. All of the usage notes for **pwrite()** apply to **pwrite64()**. See *Usage Notes* in the **pwrite** API.

Example

The following example writes a specific number of bytes to a file:

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

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

#define mega_string_len 1000000

main() {
    char *mega_string;
    int file_descriptor;
    int ret;
    off64_t off=5;
    char fn[]="write.file";

    if ((mega_string = (char*) malloc(mega_string_len+off)) == NULL)
        perror("malloc() error");
    else if ((file_descriptor = creat64(fn, S_IWUSR)) < 0)
        perror("creat64() error");
    else {
        memset(mega_string, '0', mega_string_len);
        if ((ret = pwrite64(file_descriptor, mega_string, mega_string_len, off))
== -1)
            perror("pwrite64() error");
        else printf("pwrite64() wrote %d bytes at offset %d\n", ret, off);
        if (close(file_descriptor)!= 0)
            perror("close() error");
        if (unlink(fn)!= 0)
            perror("unlink() error");
    }
}

```

Output:

```
pwrite64() wrote 1000000 bytes at offset 5
```



API introduced: V5R2

[Top](#) | [UNIX-Type APIs](#) | [APIs by category](#)

Integrated File System APIs--Time Stamp Updates

Each object (file and directory) has three time values associated with it:

Access Time The time that the data in the object is accessed.

Change Time The time that the attributes of the object are changed.

Modify Time The time that the data in the object is changed.

These values are returned by the **stat()**, **fstat()**, **lstat()**, **»**and **QlgStat()** APIs.

When it is stated that an API sets or updates one of these time values, the value may be "marked for update" by the API rather than actually updated. When a subsequent **stat()**, **fstat()**, **lstat()**, **»**and **QlgStat()** API is called, or the file is closed by all processes, the times that were previously "marked for update" are updated and the update marks are cleared.

The value of these times is measured in seconds since the Epoch. The Epoch is the time 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time. If the system date is set prior to 1970, all time values will be zero. The following table shows which of these times are "marked for update" by each of the APIs.

<i>Time Stamp Updates for Integrated File System APIs</i>			
Function	Access	Change	Modify
access	No	No	No
»accessx	No	No	No «
chdir	No	No	No
chmod	No	Yes	No
chown	No	Yes	No
close	No	No	No
closedir	No	No	No
creat ¹ (new file)	Yes	Yes	Yes
creat ¹ (parent directory of new file)	No	Yes	Yes
creat ² (existing file)	No	Yes	Yes
DosSetFileLocks	No	No	No
DosSetRelMaxFH	No	No	No
dup	No	No	No
dup2	No	No	No
»faccessx	No	No	No «
»fchdir	No	No	No «
fchmod	No	Yes	No
fchown	No	Yes	No
fcntl	No	No	No
fpathconf	No	No	No
fstat	No	No	No
fstatvfs	No	No	No

fsync	No	No	No
ftruncate	No	Yes	Yes
getcwd	Yes ³	No	No
getegid	No	No	No
geteuid	No	No	No
getgid	No	No	No
getgrgid	No	No	No
getgrgid_r	No	No	No
getgrnam	No	No	No
getgrnam_r	No	No	No
getgroups	No	No	No
getpwnam	No	No	No
getpwnam_r	No	No	No
getpwuid	No	No	No
getpwuid_r	No	No	No
getuid	No	No	No
givedescriptor	No	No	No
ioctl	No	No	No
lchown	No	Yes	No
link ⁴ (file)	No	Yes	No
link ⁴ (parent directory)	No	Yes	Yes
lseek	No	No	No
lstat	No	No	No
mkdir ⁵ (new directory)	Yes	Yes	Yes
mkdir ⁵ (parent directory)	No	Yes	Yes
mkfifo ⁶ (new directory)	Yes	Yes	Yes
mkfifo ⁶ (parent directory)	No	Yes	Yes
open O_CREAT ⁷ (new file)	Yes	Yes	Yes
open O_CREAT ⁷ (parent directory)	No	Yes	Yes
open O_TRUNC ⁸ (existing file)	No	Yes	Yes
open ⁹ (existing file)	No	No	No
opendir	No	No	No
pathconf	No	No	No
»pread	Yes	No	No «
»pread64	Yes	No	No «
»pwrite	No	Yes	Yes «
»pwrite64	No	Yes	Yes «
QlgAccess	No	No	No
»QlgAccessx	No	No	No «
QlgChdir	No	No	No
QlgChmod	No	Yes	No

QlgChown	No	Yes	No
QlgCreat ¹ (new file)	Yes	Yes	Yes
QlgCreat ¹ (parent directory of new file)	No	Yes	Yes
QlgCreat ² (existing file)	No	Yes	Yes
QlgCvtPathToQSYSObjName	No	No	No
QlgGetAttr	No	Yes	No
QlgGetcwd	Yes ³	No	No
QlgGetPathFromFileID	Yes ¹⁰	No	No
QlgLchown	No	Yes	No
QlgLink ⁴ (file)	No	Yes	No
QlgLink ⁴ (parent directory)	No	Yes	Yes
QlgLstat	No	No	No
QlgMkdir ⁵ (new directory)	Yes	Yes	Yes
QlgMkdir ⁵ (parent directory)	No	Yes	Yes
QlgMkfifo ⁵ (new directory)	Yes	Yes	Yes
QlgMkfifo ⁵ (parent directory)	No	Yes	Yes
QlgOpen O_CREAT ⁷ (new file)	Yes	Yes	Yes
QlgOpen O_CREAT ⁷ (parent directory)	No	Yes	Yes
QlgOpen O_TRUNC ⁸ (existing file)	No	Yes	Yes
QlgOpen ⁹ (existing file)	No	No	No
QlgOpendir	No	No	No
QlgPathconf	No	No	No
QlgProcessSubtree	Yes	No	No
QlgReaddir	Yes	No	No
QlgReaddir_r	Yes	No	No
QlgReadlink	Yes	No	No
QlgRenameKeep (parent directories)	No	Yes	Yes
QlgRenameUnlink (parent directories)	No	Yes	Yes
QlgRmdir (parent directory)	No	Yes	Yes
QlgSetAttr	No	Yes	No
QlgStat	No	No	No
QlgStatvfs	No	No	No
QlgSymlink ¹¹ (new link)	Yes	Yes	Yes
QlgSymlink ¹¹ (parent directory)	No	Yes	Yes
QlgUtime ¹³	No	Yes	No
QlgUnlink ¹² (file)	No	Yes	No
QlgUnlink ¹² (parent directory)	No	Yes	Yes
❖QP0FPTOS	Yes	No	No ❖
Qp0lCvtPathToQSYSObjName	No	No	No
Qp0lGetAttr	No	Yes	No
Qp0lGetPathFromFileID	Yes ¹⁰	No	No

Qp0lProcessSubtree	Yes	No	No
Qp0lRenameKeep (parent directories)	No	Yes	Yes
Qp0lRenameUnlink (parent directories)	No	Yes	Yes
❖QP0LROR	No	No	No ❖
Qp0lSetAttr	No	Yes	No
qsysetegid()	No	No	No
qsyseteuid()	No	No	No
qsysetgid()	No	No	No
qsysetregid()	No	No	No
qsysetreuid()	No	No	No
qsysetuid()	No	No	No
read	Yes	No	No
readv	Yes	No	No
readdir	Yes	No	No
readdir_r	Yes	No	No
readlink	Yes	No	No
rewinddir	No	No	No
rmdir (parent directory)	No	Yes	Yes
select	No	No	No
stat	No	No	No
statvfs	No	No	No
symlink ¹¹ (new link)	Yes	Yes	Yes
symlink ¹¹ (parent directory)	No	Yes	Yes
sysconf	No	No	No
takedescriptor	No	No	No
umask	No	No	No
unlink ¹² (file)	No	Yes	No
unlink ¹² (parent directory)	No	Yes	Yes
utime ¹³	No	Yes	No
write	No	Yes	Yes
writev	No	Yes	Yes

Notes:

1. When the file did not previously exist, a successful **creat()** or **QlgCreat()** set the access, change, and modification times for the new file. It also sets the change and modification times of the directory that contains the new file (parent directory).
2. When the file previously existed, a successful **creat()** or **QlgCreat()** sets the change and modification times for the file.
3. The access time of each directory in the absolute path name of the current directory (excluding the current directory itself) is updated.
4. A successful **link()** or **QlgLink()** sets the change time of the file and the change and modification times of the directory that contains the new link (parent directory).
5. A successful **mkdir()** or **QlgMkdir()** sets the access, change, and modification times for the new directory. It also sets the change and modification times of the directory that contains the new directory (parent directory).
6. A successful **mkfifo()** or **QlgMkfifo()** sets the access, change, and modification times for the new FIFO (first-in-first-out) special file. It also sets the change and modification times of the parent directory that contains the new FIFO file.
7. When **O_CREAT** is specified and the file did not previously exist, a successful **open()** or **QlgOpen()** sets the access, change, and modification times for the new file. It also sets the change and modification times of the directory that contains the new file (parent directory).
8. When **O_TRUNC** is specified and the file previously existed, a successful **open()** or **QlgOpen()** sets the change and modification times for the file.
9. When **O_CREAT** and **O_TRUNC** are not specified, **open()** or **QlgOpen()** does not update any time stamps.
10. A successful **Qp0lGetPathFromFileID()** or **QlgGetPathFromFileID()** sets the access time of each directory in the absolute path name to the file.
11. A successful **symlink()** or **QlgSymlink()** sets the access, change, and modification times for the new symbolic link. It also sets the change and modification times of the directory that contains the new directory (parent directory).
12. A successful **unlink()** or **QlgUnlink()** sets the change and modification times of the directory that contains the file being unlinked (parent directory). If the link count for the file is not zero, the change time for the file is set.
13. A successful **utime()** or **QlgUtime()** sets the access and modify times of the file as specified by the application. The change time of the file is set to the current time.

Header Files for UNIX-Type Functions

Programs using the UNIX-type functions must include one or more header files that contain information needed by the functions, such as:

- Macro definitions
- Data type definitions
- Structure definitions
- Function prototypes

The header files are provided in the QSYSINC library, which is optionally installable. Make sure QSYSINC is on your system before compiling programs that use these header files. For information on installing the QSYSINC library, see [Data structures and the QSYSINC Library](#).

The table below shows the file and member name in the QSYSINC library for each header file used by the UNIX-type APIs in this publication.

Name of Header File	Name of File in QSYSINC	Name of Member
arpa/inet.h	ARPA	INET
arpa/nameser.h	ARPA	NAMESER
bse.h	H	BSE
bsedos.h	H	BSEDOS
bseerr.h	H	BSEERR
dirent.h	H	DIRENT
errno.h	H	ERRNO
fcntl.h	H	FCNTL
grp.h	H	GRP
»inttypes.h	H	INTTYPES«
limits.h	H	LIMITS
»mman.h	H	MMAN«
netdbh.h	H	NETDB
»netinet/icmp6.h	NETINET	ICMP6«
net/if.h	NET	IF
netinet/in.h	NETINET	IN
netinet/ip_icmp.h	NETINET	IP_ICMP
netinet/ip.h	NETINET	IP
»netinet/ip6.h	NETINET	IP6«
netinet/tcp.h	NETINET	TCP
netinet/udp.h	NETINET	UDP
netns/idp.h	NETNS	IDP
netns/ipx.h	NETNS	IPX
netns/ns.h	NETNS	NS
netns/sp.h	NETNS	SP
net/route.h	NET	ROUTE
nettel/tel.h	NETTEL	TEL

os2.h	H	OS2
os2def.h	H	OS2DEF
pwd.h	H	PWD
Qlg.h	H	QLG
qp0lflop.h	H	QP0LFLOP
»qp0ljrnl.h	H	QP0LJRNL«
»qp0lrnr.h	H	QP0LROR«
Qp0lstdi.h	H	QP0LSTDI
qp0wpid.h	H	QP0WPID
qp0zdipc.h	H	QP0ZDIPC
qp0zipc.h	H	QP0ZIPC
qp0zolip.h	H	QP0ZOLIP
qp0zolsm.h	H	QP0ZOLSM
qp0zripc.h	H	QP0ZRIPC
qp0ztrc.h	H	QP0ZTRC
qp0ztrml.h	H	QP0ZTRML
qp0z1170.h	H	QP0Z1170
»qsoasync.h	H	QSOASYNC«
qtnxaapi.h	H	QTNXAAPI
qtnxadtp.h	H	QTNXADTP
qtomeapi.h	H	QTOMEAPI
qtossapi.h	H	QTOSSAPI
resolv.h	H	RESOLVE
semaphore.h	H	SEMAPHORE
signal.h	H	SIGNAL
spawn.h	H	SPAWN
ssl.h	H	SSL
sys/errno.h	H	ERRNO
sys/ioctl.h	SYS	IOCTL
sys/ipc.h	SYS	IPC
sys/layout.h	H	LAYOUT
sys/limits.h	H	LIMITS
sys/msg.h	SYS	MSG
sys/param.h	SYS	PARAM
»sys/resource.h	SYS	RESOURCE«
sys/sem.h	SYS	SEM
sys/setjmp.h	SYS	SETJMP
sys/shm.h	SYS	SHM
sys/signal.h	SYS	SIGNAL
sys/socket.h	SYS	SOCKET
sys/stat.h	SYS	STAT
sys/statvfs.h	SYS	STATVFS

sys/time.h	SYS	TIME
sys/types.h	SYS	TYPES
sys/uio.h	SYS	UIO
sys/un.h	SYS	UN
sys/wait.h	SYS	WAIT
» ulimit.h	H	ULIMIT «
unistd.h	H	UNISTD
utime.h	H	UTIME

You can display a header file in QSYSINC by using one of the following methods:

- Using your editor. For example, to display the **unistd.h** header file using the Source Entry Utility editor, enter the following command:

```
STRSEU SRCFILE(QSYSINC/H) SRCMBR(UNISTD) OPTION(5)
```

- Using the Display Physical File Member command. For example, to display the **sys/stat.h** header file, enter the following command:

```
DSPPFM FILE(QSYSINC/SYS) MBR(STAT)
```

You can print a header file in QSYSINC by using one of the following methods:

- Using your editor. For example, to print the **unistd.h** header file using the Source Entry Utility editor, enter the following command:

```
STRSEU SRCFILE(QSYSINC/H) SRCMBR(UNISTD) OPTION(6)
```

- Using the Copy File command. For example, to print the **sys/stat.h** header file, enter the following command:

```
CPYF FROMFILE(QSYSINC/SYS) TOFILE(*PRINT) FROMMBR(STAT)
```

Symbolic links to these header files are also provided in directory /QIBM/include.

Errno Values for UNIX-Type Functions

Programs using the UNIX-type functions may receive error information as *errno* values. The possible values returned are listed here in ascending *errno* value sequence.

Name	Value	Text
EDOM	3001	A domain error occurred in a math function.
ERANGE	3002	A range error occurred.
ETRUNC	3003	Data was truncated on an input, output, or update operation.
ENOTOPEN	3004	File is not open.
ENOTREAD	3005	File is not opened for read operations.
EIO	3006	Input/output error.
ENODEV	3007	No such device.
ERECIO	3008	Cannot get single character for files opened for record I/O.
ENOTWRITE	3009	File is not opened for write operations.
ESTDIN	3010	The stdin stream cannot be opened.
ESTDOUT	3011	The stdout stream cannot be opened.
ESTDERR	3012	The stderr stream cannot be opened.
EBADSEEK	3013	The positioning parameter in fseek is not correct.
EBADNAME	3014	The object name specified is not correct.
EBADMODE	3015	The type variable specified on the open function is not correct.
EBADPOS	3017	The position specifier is not correct.
ENOPOS	3018	There is no record at the specified position.
ENUMMBRS	3019	Attempted to use ftell on multiple members.
ENUMRECS	3020	The current record position is too long for ftell.
EINVAL	3021	The value specified for the argument is not correct.
EBADFUNC	3022	Function parameter in the signal function is not set.
ENOENT	3025	No such path or directory.
ENOREC	3026	Record is not found.
EPERM	3027	The operation is not permitted.
EBADDATA	3028	Message data is not valid.
EBUSY	3029	Resource busy.
EBADOPT	3040	Option specified is not valid.
ENOTUPD	3041	File is not opened for update operations.
ENOTDLT	3042	File is not opened for delete operations.

EPAD	3043	The number of characters written is shorter than the expected record length.
EBADKEYLN	3044	A length that was not valid was specified for the key.
EPUTANDGET	3080	A read operation should not immediately follow a write operation.
EGETANDPUT	3081	A write operation should not immediately follow a read operation.
EIOERROR	3101	A nonrecoverable I/O error occurred.
EIORECERR	3102	A recoverable I/O error occurred.
EACCES	3401	Permission denied.
ENOTDIR	3403	Not a directory.
ENOSPC	3404	No space is available.
EXDEV	3405	Improper link.
EAGAIN	3406	Operation would have caused the process to be suspended.
EWOULDBLOCK	3406	Operation would have caused the process to be suspended.
EINTR	3407	Interrupted function call.
EFAULT	3408	The address used for an argument was not correct.
ETIME	3409	Operation timed out.
ENXIO	3415	No such device or address.
EAPAR	3418	Possible APAR condition or hardware failure.
ERECURSE	3419	Recursive attempt rejected.
EADDRINUSE	3420	Address already in use.
EADDRNOTAVAIL	3421	Address is not available.
EAFNOSUPPORT	3422	The type of socket is not supported in this protocol family.
EALREADY	3423	Operation is already in progress.
ECONNABORTED	3424	Connection ended abnormally.
ECONNREFUSED	3425	A remote host refused an attempted connect operation.
ECONNRESET	3426	A connection with a remote socket was reset by that socket.
EDESTADDRREQ	3427	Operation requires destination address.
EHOSTDOWN	3428	A remote host is not available.
EHOSTUNREACH	3429	A route to the remote host is not available.
EINPROGRESS	3430	Operation in progress.
EISCONN	3431	A connection has already been established.
EMSGSIZE	3432	Message size is out of range.
ENETDOWN	3433	The network currently is not available.
ENETRESET	3434	A socket is connected to a host that is no longer available.

ENETUNREACH	3435	Cannot reach the destination network.
ENOBUFS	3436	There is not enough buffer space for the requested operation.
ENOPROTOPT	3437	The protocol does not support the specified option.
ENOTCONN	3438	Requested operation requires a connection.
ENOTSOCK	3439	The specified descriptor does not reference a socket.
ENOTSUP	3440	Operation is not supported.
EOPNOTSUPP	3440	Operation is not supported.
EPFNOSUPPORT	3441	The socket protocol family is not supported.
EPROTONOSUPPORT	3442	No protocol of the specified type and domain exists.
EPROTOTYPE	3443	The socket type or protocols are not compatible.
ERCVDERR	3444	An error indication was sent by the peer program.
ESHUTDOWN	3445	Cannot send data after a shutdown.
ESOCKTNOSUPPORT	3446	The specified socket type is not supported.
ETIMEDOUT	3447	A remote host did not respond within the timeout period.
EUNATCH	3448	The protocol required to support the specified address family is not available at this time.
EBADF	3450	Descriptor is not valid.
EMFILE	3452	Too many open files for this process.
ENFILE	3453	Too many open files in the system.
EPIPE	3455	Broken pipe.
ECANCEL	3456	Operation cancelled.
EEXIST	3457	File exists.
EDEADLK	3459	Resource deadlock avoided.
ENOMEM	3460	Storage allocation request failed.
EOWNERTERM	3462	The synchronization object no longer exists because the owner is no longer running.
EDESTROYED	3463	The synchronization object was destroyed, or the object no longer exists.
ETERM	3464	Operation was terminated.
ENOENT1	3465	No such file or directory.
ENOEQFLOG	3466	Object is already linked to a dead directory.
EEMPTYDIR	3467	Directory is empty.
EMLINK	3468	Maximum link count for a file was exceeded.

ESPIPE	3469	Seek request is not supported for object.
ENOSYS	3470	Function not implemented.
EISDIR	3471	Specified target is a directory.
EROFS	3472	Read-only file system.
EUNKNOWN	3474	Unknown system state.
EITERBAD	3475	Iterator is not valid.
EITERSTE	3476	Iterator is in wrong state for operation.
EHRICLSBAD	3477	HRI class is not valid.
EHRICLBAD	3478	HRI subclass is not valid.
EHRITYPBAD	3479	HRI type is not valid.
ENOTAPPL	3480	Data requested is not applicable.
EHRIREQTYP	3481	HRI request type is not valid.
EHRINAMEBAD	3482	HRI resource name is not valid.
EDAMAGE	3484	A damaged object was encountered.
ELOOP	3485	A loop exists in the symbolic links.
ENAMETOOLONG	3486	A path name is too long.
ENOLCK	3487	No locks are available.
ENOTEMPTY	3488	Directory is not empty.
ENOSYSRSC	3489	System resources are not available.
ECONVERT	3490	Conversion error.
E2BIG	3491	Argument list is too long.
EILSEQ	3492	Conversion stopped due to input character that does not belong to the input codeset.
ETYPE	3493	Object type mismatch.
EBADDIR	3494	Attempted to reference a directory that was not found or was destroyed.
EBADOBJ	3495	Attempted to reference an object that was not found, was destroyed, or was damaged.
EIDXINVAL	3496	Data space index used as a directory is not valid.
ESOFTDAMAGE	3497	Object has soft damage.
ENOTENROLL	3498	User is not enrolled in system distribution directory.
EOffline	3499	Object is suspended.
EROOBJ	3500	Object is a read-only object.
EEAHDDSI	3501	Hard damage on extended attribute data space index.
EEASDDSI	3502	Soft damage on extended attribute data space index.
EEAHDDS	3503	Hard damage on extended attribute data space.
EEASDDS	3504	Soft damage on extended attribute data space.
EEADUPRC	3505	Duplicate extended attribute record.

ELOCKED	3506	Area being read from or written to is locked.
EFBIG	3507	Object too large.
EIDRM	3509	The semaphore, shared memory, or message queue identifier is removed from the system.
ENOMSG	3510	The queue does not contain a message of the desired type and (msgflg logically ANDed with IPC_NOWAIT).
EFILECVT	3511	File ID conversion of a directory failed.
EBADFID	3512	A file ID could not be assigned when linking an object to a directory.
ESTALE	3513	File handle was rejected by server.
ESRCH	3515	No such process.
ENOTSIGINIT	3516	Process is not enabled for signals.
ECHILD	3517	No child process.
EBADH	3520	Handle is not valid.
ETOOMANYREFS	3523	The operation would have exceeded the maximum number of references allowed for a descriptor.
ENOTSAFE	3524	Function is not allowed.
E_OVERFLOW	3525	Object is too large to process.
EJRNDDAMAGE	3526	Journal is damaged.
EJRNINACTIVE	3527	Journal is inactive.
EJRNRCVSPC	3528	Journal space or system storage error.
EJRNRMNT	3529	Journal is remote.
ENEWJRNRCV	3530	New journal receiver is needed.
ENEWJRN	3531	New journal is needed.
EJOURNALED	3532	Object already journaled.
EJRNENTTOOLONG	3533	Entry is too large to send.
EDATALINK	3534	Object is a datalink object.
ENOTAVAIL	3535	IASP is not available.
ENOTTY	3536	I/O control operation is not appropriate.
EFBIG2	3540	Attempt to write or truncate file past its sort file size limit.
ETXTBSY	3543	Text file busy.
EASPGRPNOTSET	3544	ASP group not set for thread.
ERESTART	3545	A system call was interrupted and may be restarted.