UNIX-Type APIs (V5R2)

Integrated File System (IFS) APIs

Volume 2 -- QlgAccess() through writev() and Process a Path Name Exit Program

---

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 prwrite64(); 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

- QlgAccess() (Determine file accessibility (using NLS-enabled path name))
- QlgAccessx() (Determine File Accessibility for a Class of Users (using NLS-enabled path name))
- QlgChdir() (Change current directory (using NLS-enabled path name))
- QlgChmod() (Change file authorizations (using NLS-enabled path name))
- QlgChown() (Change owner and group of file (using NLS-enabled path name))
- QlgCreat() (Create or rewrite file (using NLS-enabled path name))
- QlgCreat64() (Create or rewrite a file (large file enabled and using NLS-enabled path name))
- QlgCvtPathToQSYSObjName() (Resolve integrated file system path name into QSYS object name (using NLS-enabled path name))
- QlgGetAttr() (Get attributes (using NLS-enabled path name))
- QlgGetcwd() (Get current directory (using NLS-enabled path name))
- QlgGetPathFromFileID() (Get path name of object from its file ID (using NLS-enabled path name))
- QlgGetpwnam() (Get user information for user name (using NLS-enabled path name))
- QlgGetpwnam_r() (Get user information for user name (using NLS-enabled path name))
- QlgGetpwuid() (Get user information for user ID (using NLS-enabled path name))
- QlgGetpwuid_r() (Get user information for user ID (using NLS-enabled path name))
- QlgLchown() (Change owner and group of symbolic link (using NLS-enabled path name))
- QlgLink() (Create link to file (using NLS-enabled path name))
- QlgLstat() (Get file or link information (using NLS-enabled path name))
- QlgLstat64() (Get file or link information (large file enabled and using NLS-enabled path name))
- QlgMkdir() (Make directory (using NLS-enabled path name))
- QlgMkfifo() (Make FIFO special file (using NLS-enabled path name))
- QlgOpen() (Open a file (using NLS-enabled path name))

- QlgOpen64() (Open file (large file enabled and using NLS-enabled path name))
- QlgOpendir() (Open directory (using NLS-enabled path name))
- QlgPathconf() (Get configurable path name variables (using NLS-enabled path name))
- QlgProcessSubtree() (Process a path name (using NLS-enabled path name))
- QlgReaddir() (Read directory entry (using NLS-enabled path name))
- QlgReaddir_r() (Read directory entry (using NLS-enabled path name))
- QlgReadlink() (Read value of symbolic link (using NLS-enabled path name))
- QlgRenameKeep() (Rename file or directory, keep "new" if it exists (using NLS-enabled path name))
- QlgRenameUnlink() (Rename file or directory, unlink "new" if it exists (using NLS-enabled path name))
- QlgRmdir() (Remove directory (using NLS-enabled path name))
- QlgSaveStgFree() (Save Storage Free (using NLS-enabled path name))
- QlgSetAttr() (Set attributes (using NLS-enabled path name))
- QlgStat() (Get file information (using NLS-enabled path name))
- QlgStat64() (Get file information (large file enabled and using NLS-enabled path name))
- QlgStatvfs() (Get file system information (using NLS-enabled path name))
- QlgStatvfs64() (Get file system information (64-bit enabled and using NLS-enabled path name))
- QlgSymlink() (Make symbolic link (using NLS-enabled path name))
- QlgUnlink() (Remove link to file (using NLS-enabled path name))
- QlgUtime() (Set file access and modification times (using NLS-enabled path name))
- QP0FPTOS (Perform Miscellaneous File System Functions)
- Qp0lCvtPathToQSYSObjName() (Resolve integrated file system path name into QSYS object name)
- QP0LFLOP (Perform file system operation)
- Qp0lGetAttr() (Get attributes)
- Qp0lGetPathFromFileID() (Get path name of object from its file ID)
- Qp0lOpen() (Open file)
- Qp0lProcessSubtree() (Process a path name)
- Qp0lRenameKeep() (Rename file or directory, keep *new* if it exists)
- Qp0lRenameUnlink() (Rename file or directory, unlink *new* if it exists)
- QP0LROR (Retrieve Object References)
- Qp0lSaveStgFree() (Save Storage Free)
- Qp0lSetAttr() (Set attributes)
- Qp0lUnlink() (Remove link to file)
- Qp0zPipe() (Create interprocess channel with sockets)
- qsygetgroups() (Get Supplemental Group IDs)

- qsysetegid() (Set effective group ID)
- qsyseteuid() (Set effective user ID)
- qsysetgid() (Set group ID)
- qsysetgroups() (Set Supplemental Group IDs)
- qsysetregid() (Set real and effective group IDs)
- qsysetreuid() (Set real and effective user IDs)
- qsysetuid() (Set user ID)
- QZNFRTVE (Retrieve network file system export entries)
- read() (Read from Descriptor)
- readdir() (Read directory entry)
- readdir_r() (Read directory entry)
- readdir_r_ts64() (Read directory entry)
- readlink() (Read value of symbolic link)
- readv() (Read from Descriptor Using Multiple Buffers)
- rename() (Rename file or directory)
- rewinddir() (Reset directory stream)
- rmdir() (Remove directory)
- stat() (Get file information)
- stat64() (Get file information (large file enabled))
- statvfs() (Get file system information)
- statvfs64() (Get file system information (large file enabled))
- symlink() (Make symbolic link)
- sysconf() (Get system configuration variables)
- umask() (Set authorization mask for job)
- unlink() (Remove link to file)
- utime() (Set file access and modification times)
- write() (Write to Descriptor)
- writev() (Write to Descriptor Using Multiple Buffers)

Exit programs

- Process a Path Name
- Save Storage Free

Integrated File System APIs--Time Stamp Updates
Header Files for UNIX-Type Functions
Errno Values for UNIX-Type Functions

Integrated File System APIs

QlgAccess() through writev() and Process a Path Name Exit Program

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 (QlgAccess() through writev() and Process a Path Name Exit Program) are:

- QlgAccess() (Determine file accessibility (using NLS-enabled path name)) determines whether a file can be accessed in a particular manner.
- ≫QlgAccessx() (Determine File Accessibility for a Class of Users (using NLS-enabled path name)) determines whether a file can be accessed in a particular manner by a specified class of users.≪
- QlgChdir() (Change current directory (using NLS-enabled path name)) makes the directory named by path the new current directory.
- QlgChmod() (Change file authorizations (using NLS-enabled path name)) changes the mode of the file or directory specified in path.
- QlgChown() (Change owner and group of file (using NLS-enabled path name)) changes the owner and group of a file.
- QlgCreat() (Create or rewrite file (using NLS-enabled path name)) creates a new file or rewrites an existing file so that it is truncated to zero length.
- QlgCreat64() (Create or rewrite a file (large file enabled and using NLS-enabled path name)) creates a new file or rewrites an existing file so that it is truncated to zero length.
- QlgCvtPathToQSYSObjName() (Resolve integrated file system path name into QSYS object name (using NLS-enabled path name)) resolves a given integrated file system path name into the three-part QSYS.LIB file system name: library, object, and member.
- QlgGetAttr() (Get attributes (using NLS-enabled path name)) gets one or more attributes, on a single call, for the object that is referred to by the input Path_Name.
- QlgGetcwd() (Get current directory (using NLS-enabled path name)) determines the absolute path name of the current directory and returns a pointer to it.
- QlgGetPathFromFileID() (Get path name of object from its file ID (using NLS-enabled path name)) determines an absolute path name of the file identified by fileid and stores it in buf.
- QlgGetpwnam() (Get user information for user name (using NLS-enabled path name)) returns a pointer to an object of type struct qplg_passwd containing an entry from the user database with a matching name.
- QlgGetpwnam_r() (Get user information for user name (using NLS-enabled path name)) updates the qplg_passwd structure pointed to by pwd and stores a pointer to that structure in the location pointed to by result.
- QlgGetpwuid() (Get user information for user ID (using NLS-enabled path name)) returns a pointer to an object of type struct qplg_passwd containing an entry from the user database with a matching user ID (UID).
- QlgGetpwuid_r() (Get user information for user ID (using NLS-enabled path name)) updates the qplg_passwd structure pointed to by pwd and stores a pointer to that structure in the location pointed to by result.
- QlgLchown() (Change owner and group of symbolic link (using NLS-enabled path name)) changes

the owner and group of a file.

- QlgLink() (Create link to file (using NLS-enabled path name)) 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.
- QlgLstat() (Get file or link information (using NLS-enabled path name)) gets status information about a specified file and places it in the area of memory pointed to by buf.
- QlgLstat64() (Get file or link information (large file enabled and using NLS-enabled path name)) gets status information about a specified file and places it in the area of memory pointed to by buf.
- QlgMkdir() (Make directory (using NLS-enabled path name)) creates a new, empty directory whose name is defined by path.
- QlgMkfifo() (Make FIFO special file (using NLS-enabled path name)) creates a new FIFO special file whose name is defined by path.
- QlgOpen() (Open a file (using NLS-enabled path name)) opens a file or creates a new, empty file whose name is defined by path and returns a number called a file descriptor.
- QlgOpen64() (Open file (large file enabled and using NLS-enabled path name)) opens a file and returns a number called a file descriptor.
- QlgOpendir() (Open directory (using NLS-enabled path name)) opens a directory so it can be read.
- QlgPathconf() (Get configurable path name variables (using NLS-enabled path name)) lets an application determine the value of a configuration variable (name) associated with a particular file or directory (path).
- QlgProcessSubtree() (Process a path name (using NLS-enabled path name)) searches the directory tree under a specific path name.
- QlgReaddir() (Read directory entry (using NLS-enabled path name)) returns a pointer to a structure describing the next directory entry in the directory stream associated with dirp.
- QlgReaddir_r() (Read directory entry (using NLS-enabled path name)) initializes a structure that is referenced by entry to represent the next directory entry in the directory stream that is associated with dirp.
- QlgReadlink() (Read value of symbolic link (using NLS-enabled path name)) places the contents of the symboliclink path in the buffer buf.
- QlgRenameKeep() (Rename file or directory, keep "new" if it exists (using NLS-enabled path name)) renames a file or a directory specified by old to the name given by new.
- QlgRenameUnlink() (Rename file or directory, unlink "new" if it exists (using NLS-enabled path name)) renames a file or a directory specified by old to the name given by new.
- QlgRmdir() (Remove directory (using NLS-enabled path name)) removes a directory, path, provided that the directory is empty; that is, the directory contains no entries other than 'dot' (.) or 'dot-dot' (..).
- QlgSaveStgFree() (Save Storage Free (using NLS-enabled path name)) calls a user-supplied exit program to save an *STMF iSeries object type and, upon successful completion of the exit program, frees the storage for the object and marks the object as storage freed.
- QlgSetAttr() (Set attributes (using NLS-enabled path name)) sets one of a set of defined attributes, on each call, for the object that is referred to by the input *Path_Name.
- QlgStat() (Get file information (using NLS-enabled path name)) gets status information about a specified file and places it in the area of memory pointed to by the buf argument.
- QlgStat64() (Get file information (large file enabled and using NLS-enabled path name)) gets status information about a specified file and places it in the area of memory pointed to by the buf argument.

- QlgStatvfs() (Get file system information (using NLS-enabled path name)) gets status information about the file system that contains the file named by the path argument.
- QlgStatvfs64() (Get file system information (64-bit enabled and using NLS-enabled path name)) gets status information about the file system that contains the file named by the path argument.
- QlgSymlink() (Make symbolic link (using NLS-enabled path name)) creates the symbolic link named by slink with the value specified by pname.
- QlgUnlink() (Remove link to file (using NLS-enabled path name)) removes a directory entry that refers to a file.
- QlgUtime() (Set file access and modification times (using NLS-enabled path name)) sets the access and modification times of path to the values in the utimbuf structure.
- »QP0FPTOS (Perform Miscellaneous File System Functions) performs a variety of file system functions.«
- Qp0lCvtPathToQSYSObjName() (Resolve integrated file system path name into QSYS object name) resolves a given integrated file system path name into the three-part QSYS.LIB file system name: library, object, and member.
- QP0LFLOP (Perform file system operation) performs miscellaneous file system operations.
- Qp0lGetAttr() (Get attributes) gets one or more attributes, on a single call, for the object that is referred to by the input Path_Name.
- Qp0lGetPathFromFileID() (Get path name of object from its file ID) determines an absolute path name of the file identified by fileid and stores it in buf.
- Qp0lOpen() (Open file) opens a file and returns a number called a file descriptor.
- Qp0lProcessSubtree() (Process a path name) searches the directory tree under a specific path name. It selects and passes objects, one at a time, to an exit program that is identified on its call. The exit program can be either a procedure or a program.
- Qp0lRenameKeep() (Rename file or directory, keep *new* if it exists) renames a file or a directory specified by old to the name given by new.
- Qp0lRenameUnlink() (Rename file or directory, unlink *new* if it exists) renames a file or a directory specified by old to the name given by new.
- »QP0LROR (Retrieve Object References) retrieves information about Integrated File System references on an object.«
- Qp0lSaveStgFree() (Save Storage Free) calls a user-supplied exit program to save an *STMF iSeries object type and, upon successful completion of the exit program, frees the storage for the object and marks the object as storage freed.
- Qp0lSetAttr() (Set attributes) renames a file or a directory specified by old to the name given by new.
- Qp0lUnlink() (Remove link to file) removes a directory entry that refers to a file.
- Qp0zPipe() (Create interprocess channel with sockets) creates a data pipe that can be used by two processes.
- »qsygetgroups() (Get Supplemental Group IDs) returns the supplemental group IDs associated with the calling thread.«
- qsysetegid() (Set effective group ID) sets the effective group ID to gid.
- qsyseteuid() (Set effective user ID) sets the effective user ID to uid.
- qsysetgid() (Set group ID) sets the real, effective and saved groups to gid.

- »qsysetgroups() (Set Supplemental Group IDs) sets the supplementary group IDs of the calling thread.«
- qsysetregid() (Set real and effective group IDs) is used to set the real and effective group IDs. The real and effective group IDs may be set to different values in the same call.
- qsysetreuid() (Set real and effective user IDs) sets the real and effective user IDs to the values specified by ruid and euid.
- qsysetuid() (Set user ID) sets the real, effective, and saved user ID to uid.
- QZNFRTVE (Retrieve network file system export entries) returns the list of Network File System (NFS) export entries for objects currently exported to NFS clients or for objects referenced in the /etc/exports file.
- read() (Read from Descriptor) reads nbyte bytes of input into the memory area indicated by buf.
- readdir() (Read directory entry) returns a pointer to a dirent structure describing the next directory entry in the directory stream associated with dirp.
- readdir_r() (Read directory entry) initializes the dirent structure that is referenced by entry to represent the next directory entry in the directory stream that is associated with dirp.
- readdir_r_ts64() (Read directory entry) initializes the dirent structure that is referenced by entry to represent the next directory entry in the directory stream that is associated with dirp.
- readlink() (Read value of symbolic link) places the contents of the symbolic link path in the buffer buf.
- readv() (Read from Descriptor Using Multiple Buffers) is used to receive data from a file or socket descriptor.
- rename() (Rename file or directory) can be used to rename a file or directory with the semantics of Qp0lRenameUnlink() or Qp0lRenameKeep().
- rewinddir() (Reset directory stream) 'rewinds' the position of an open directory stream to the beginning.
- rmdir() (Remove directory) removes a directory, path, provided that the directory is empty; that is, the directory contains no entries other than 'dot' (.) or 'dot-dot' (..).
- stat() (Get file information) gets status information about a specified file and places it in the area of memory pointed to by the buf argument.
- stat64() (Get file information (large file enabled)) gets status information about a specified file and places it in the area of memory pointed to by the buf argument.
- statvfs() (Get file system information) gets status information about the file system that contains the file named by the path argument.
- statvfs64() (Get file system information (large file enabled)) gets status information about the file system that contains the file named by the path argument.
- symlink() (Make symbolic link) creates the symbolic link named by slink with the value specified by pname.
- sysconf() (Get system configuration variables) returns the value of a system configuration option.
- umask() (Set authorization mask for job) changes the value of the file creation mask for the current job to the value specified in cmask.
- unlink() (Remove link to file) removes a directory entry that refers to a file.
- utime() (Set file access and modification times) sets the access and modification times of path to the values in the utimbuf structure.

- write() (Write to Descriptor) writes nbyte bytes from buf to the file or socket associated with file_descriptor.
- writev() (Write to Descriptor Using Multiple Buffers) is used to write data to a file or socket descriptor.

The integrated file system exit programs are:

- Process a Path Name is called by the Qp0lProcessSubtree() API for each object in the API's search that meets the caller's selection criteria. This exit program must be provided by the user.
- Save Storage Free is called by the Qp0lSaveStgFree() API to save an *STMF iSeries object type.

In addition to the functions above, the following functions, which are described in the Sockets APIs, also can operate on files in the integrated file system.

Other Functions that Operate on Files	
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.

QlgAccess()--Determine File Accessibility (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgAccess(const Qlg_Path_Name_T *path, int amode);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgAccess**() function, like the **access**() function, determines whether a file can be accessed in a particular manner. The difference is that the **QlgAccess()** function takes a pointer to a Qlg_Path_Name_T structure, while **access()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see access()--Determine File Accessibility.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name for the file to be checked for accessibility. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- access()--Determine File Accessiblity
- accessx()--Determine File Accessibility for Class of Users
- faccessx()--Determine File Accessibility for Class of Users
- QlgAccessx()--Determine File Accessibility for Class of Users (using NLS-enabled path name)
- QlgChmod--Change File Authorizations (using NLS-enabled path name)
- QlgStat()--Get File Information (using NLS-enabled path name)

Example

The following example determines how a file is accessed:

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

main()
{

  /*************************************************************/
  /*   Defininitons                                            */
  /*************************************************************/
#define mypath "/"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or this must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path;

  /*************************************************************/
  /*   Initialize Qlg_Path_Name_T parameters                   */
  /*************************************************************/
  memset((void*)path name, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);


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

Output:

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

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

»

QlgAccessx()--Determine File Accessibility for a Class of Users (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgAccessx(const Qlg_Path_Name_T *path, int amode, int who);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgAccessx**() function, like the **accessx**() function, determines whether a file can be accessed in a particular manner by a specified class of users. The difference is that the **QlgAccessx()** function takes a pointer to a Qlg_Path_Name_T structure, while **accessx()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see accessx()--Determine File Accessibility for a Class of Users.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name for the file to be checked for accessibility. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- access()--Determine File Accessiblity
- accessx()--Determine File Accessibility for a Class of Users
- faccessx()--Determine File Accessibility for a Class of Users
- QlgAccess()--Determine File Accessibility (using NLS-enabled path name)
- QlgChmod()--Change File Authorizations (using NLS-enabled path name)
- QlgStat()--Get File Information (using NLS-enabled path name)

Example

The following example determines how a file is accessed:

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

main()
{

  /**************************************************************/
  /*   Defininitons                                             */
  /**************************************************************/
#define mypath "/myfile"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or this must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path;

  /**************************************************************/
  /*   Initialize Qlg_Path_Name_T parameters                    */
  /**************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

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

Output:

In this example **QlgAccessx()** 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

QlgChdir()--Change Current Directory (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgChdir(const Qlg_Path_Name_T *path);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgChdir()** function, like the **chdir()** function, makes the directory named by *path* the new current directory. The difference is that the **QlgChdir()** function takes a pointer to a Qlg_Path_Name_T structure, while **chdir()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see chdir()--Change Current Directory.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the directory that should become the current directory. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- chdir()--Change Current Directory
- QlgGetcwd()--Get Current Directory (using NLS-enabled path name)
- fchdir()--Change Current Directory by Descriptor

Example

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

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

```
#include <unistd.h>

main() {
#define mypath "/tmpXXX"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or this */
                  /* this be a pointer to the path name.  */
  };
  struct pnstruct path;

  /***************************************************************/
  /*   Initialize Qlg_Path_Name_T parameters                      */
  /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if (QlgChdir((Qlg_Path_Name_T *)&path) != 0)
  {
    printf("QlgChdir() to /tmpXXX failed.");
  }
  else
  {
   printf("QlgChdir() changed the current directory ");
   printf("to '%s'.\n", mypath);
  }
}
```

Output:

```
QlgChdir() changed the current directory to '/tmpxxx'.
(or if error, such as path not found: QlgChdir() to /tmpXXX failed.)
```

API introduced: V5R1

QlgChmod()--Change File Authorizations (using NLS-enabled path name)

Syntax

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

 int QlgChmod(Qlg_Path_Name_T *path, mode_t mode);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgChmod()** function, like 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*. The difference is that the **QlgChmod()** function takes a pointer to a Qlg_Path_Name_T structure, while **chmod()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see chmod()--Change File Authorizations.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains the path name or a pointer to the path name of the file whose mode is being changed. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- chmod()--Change File Authorizations
- QlgChown()--Change Owner and Group of File (using NLS-enabled path name)
- QlgMkdir()--Make Directory (using NLS-enabled path name)
- QlgStat()--Get File Information (using NLS-enabled path name)

Example

The following example changes the permissions for a file:

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <Qp0lstdi.h>

main() {
  int file_descriptor;
  struct stat info;

 #define mypath "temp.file"
 const char US_const[3]= "US";
 const char Language_const[4] ="ENU";
 typedef struct pnstruct
 {
   Qlg_Path_Name_T qlg_struct;
   char pn[100]; /* This array size must be >= the  */
                   /* length of the path name or this must */
                   /* be a pointer to the path name.  */
 };
 struct pnstruct path;

  /***************************************************************/
  /*   Initialize Qlg_Path_Name_T parameters                     */
  /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if ((file_descriptor = QlgCreat((Qlg_Path_Name_T *)&path, S_IWUSR)) == -1)
    perror("QlgCreat() error");
  else {
    close(file_descriptor);
    QlgStat((Qlg_Path_Name_T *)&path, &info);
    printf("original permissions were: %08o\n", info.st_mode);
    if (QlgChmod((Qlg_Path_Name_T *)&path, S_IRWXU|S_IRWXG) != 0)
      perror("QlgChmod() error");
    else {
      QlgStat((Qlg_Path_Name_T *)&path, &info);
      printf("after QlgChmod(), permissions are: %08o\n", info.st_mode);
    }
    QlgUnlink((Qlg_Path_Name_T *)&path);
  }
}
```

Output:

```
original permissions were: 00100200
```

```
after QlgChmod(), permissions are: 00100770
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgChown()--Change Owner and Group of File (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgChown(Qlg_Path_Name_T *path, uid_t owner,gid_t
group);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgChown()** function, like the **chown()** function, changes the owner and group of a file. The difference is that the **QlgChown()** function takes a pointer to a Qlg_Path_Name_T structure, while **chown()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see chown()--Change Owner and Group of File.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file whose owner and group are being changed. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- chown()--Change Owner and Group of File
- QlgChmod()--Change File Authorizations (using NLS-enabled path name)
- QlgLstatu()--Get File or Link Information (using NLS-enabled path name)
- QlgStat()--Get File Information (using NLS-enabled path name)

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>
#include <fcntl.h>
#include <Qp0lstdi.h>

main() {
  int file_descriptor;
  struct stat info;

 #define mypath "temp.file"
 const char US_const[3]= "US";
 const char Language_const[4] ="ENU";
 typedef struct pnstruct
 {
   Qlg_Path_Name_T qlg_struct;
   char pn[100]; /* This array size must be >= the  */
                   /* length of the path name or this must */
                   /* be a pointer to the path name.  */
 };
 struct pnstruct path;
   /*************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                   */
   /*************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);


  if ((file_descriptor = QlgCreat((Qlg_Path_Name_T *)&path, S_IRWXU)) == -1)
    perror("creat() error");
  else {
    close(file_descriptor);
    QlgStat((Qlg_Path_Name_T *)&path, &info);
    printf("original owner was %d and group was %d\n", info.st_uid,
           info.st_gid);
    if (QlgChown((Qlg_Path_Name_T *)&path, 152, 0) != 0)
      perror("QlgChown() error");
    else {
      QlgStat((Qlg_Path_Name_T *)&path, &info);
      printf("after QlgChown(), owner is %d and group is %d\n",
             info.st_uid, info.st_gid);
    }
    QlgUnlink((Qlg_Path_Name_T *)&path);
  }
}
```

Output:

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

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgCreat()--Create or Rewrite File (using NLS-enabled path name)

Syntax

```
#include <fcntl.h>

int QlgCreat(Qlg_Path_Name_T *path, mode_t mode);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgCreat()** function, like the **creat()** function, creates a new file or rewrites an existing file so that it is truncated to zero length. The difference is that the **QlgCreat()** function takes a pointer to a Qlg_Path_Name_T structure, while **creat()** takes a pointer to a character string. See open()--Open File for more details on how the function call

```
QlgCreat(path,mode);
```

is equivalent to the call

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

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see creat()--Create or Rewrite File or open()--Open File.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file to be created or rewritten. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- creat()--Create or Rewrite File
- QlgCreat64()--Create or Rewrite a File (large file enabled and using NLS-enabled path name)

Example

The following example creates a file:

```
#include <stdio.h>
#include <fcntl.h>
#include <Qp0lstdi.h>

main() {
  char text[]="This is a test";
  int file_descriptor;
 #define mypath "creat.file"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or this must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);
  if ((file_descriptor =
        QlgCreat((Qlg_Path_Name_T *)&path, S_IRUSR | S_IWUSR)) < 0)
    perror("QlgCreat() error");
  else {
    write(file_descriptor, text, strlen(text));
    close(file_descriptor);
    QlgUnlink((Qlg_Path_Name_T *)&path);
  }
}
```

API introduced: V5R1

QlgCreat64()--Create or Rewrite a File (large file enabled and using NLS-enabled path name)

Syntax

```
#include <fcntl.h>

int QlgCreat64(Qlg_Path_Name_T *path,mode_t mode);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgCreat64()** function, like the **creat64()** function, creates a new file or rewrites an existing file so that it is truncated to zero length. The difference is that the **QlgCreat64()** function takes a pointer to a Qlg_Path_Name_T structure, while **creat64()** takes a pointer to a character string. See creat64()--Create or Rewrite a File (Large File Enabled) and open64()--Open File (Large File Enabled) for more details on how the function call

```
QlgCreat64(path,mode);
```

is equivalent to the call

```
QlgOpen64(path, O_CREAT|O_WRONLY|O_TRUNC, mode);
```

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see creat64()--Create or Rewrite a File (Large File Enabled) or open64()--Open File (Large File Enabled).

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file to be created or rewritten. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

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

Example

The following example creates a file:

```
#define _LARGE_FILE_API

#include <stdio.h>
#include <fcntl.h>
#include <Qp0lstdi.h>

main()
{
  char text[]="This is a test";
  int fd;
#define mypath "creat.file"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or this must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path;

  /*****************************************************************/
  /*   Initialize Qlg_Path_Name_T parameters                       */
  /*****************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if
   ((fd =
   QlgCreat64(
        (Qlg_Path_Name_T *)&path, S_IRUSR | S_IWUSR))
    < 0)
   {
    perror("QlgCreat64() error");
   }
  else {
    write(fd, text, strlen(text));
    close(fd);
    QlgUnlink((Qlg_Path_Name_T *)&path);
  }
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgCvtPathToQSYSObjName()-- Resolve Integrated File System Path Name into QSYS Object Name (using NLS-enabled path name)

Syntax

```
#include <qp0lstdi.h>

void QlgCvtPathToQSYSObjName(
                              Qlg_Path_Name_T *path_name,
                              void            *qsys_info,
                              char             format_name[8],
                              uint             bytes_provided,
                              uint             desired_CCSID,
                              void            *error_code);
```

Service Program Name: QP0LLIB2

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

For a description of this function and more information on the parameters, authorities required, return values, error conditions, error messages, usage notes, and related information, see Qp0lCvtPathToQSYSObjName()-- Resolve Integrated File System Path Name into QSYS Object Name.

API introduced: V5R1

QlgGetAttr()--Get Attributes (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h<
int QlgGetAttr
  (Qlg_Path_Name_T          *Path_Name,
   Qp0l_AttrTypes_List_t    *Attr_Array_ptr,
   char                     *Buffer_ptr,
   uint                      Buffer_Size_Provided,
   uint                     *Buffer_Size_Needed_ptr,
   uint                     *Num_Bytes_Returned_ptr,
   uint                      Follow_Symlnk, ...);
```

Service Program Name: QP0LLIB2

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

For a description of this function and more information on the parameters, authorities required, return values, error conditions, error messages, usage notes, related information, and an example, see Qp0lGetAttr()--Get Attributes.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgGetcwd()--Get Current Directory (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

  Qlg_Path_Name_T *QlgGetcwd(Qlg_Path_Name_T *buf,
  size_t size);
```

Service Program Name: QP0LLIB2

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgGetcwd**() function, like the **getcwd()** function, determines the absolute path name of the current directory and returns a pointer to it. The difference is that the pointer returned by **QlgGetcwd()** is a pointer to a Qlg_Path_Name_T structure that holds the absolute path name, while **getcwd()** returns a pointer to a character string or buffer that contains the null-terminated absolute path name.

Limited information on the *buf* parameter and on the *size* parameter is provided here. For more information on the parameters and for a discussion on authorities required, return values, and related information, see getcwd()--Get Current Directory.

Parameters

buf

(Output) A pointer to a Qlg_Path_Name_T structure that holds the absolute path name of the current directory. The path name is not null-terminated within the structure. For more information on the Qlg_Path_Name_T structure, see Path name format.

size

(Input) The number of bytes allocated for *buf*.

Related Information

- getcwd()--Get Current Directory
- QlgChdir()--Change Current Directory (using NLS-enabled path name)

Example

The following example determines the current directory:

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

main()
{

 #define mypath_cd "/tmp"

 const char US_const[3]= "US";
 const char Language_const[4]="ENU";
 typedef struct pnstruct
    {
     Qlg_Path_Name_T qlg_struct;
     char pn[1024];  /* This size must be large enough */
                            /* to contain the path name.     */
    };

 struct pnstruct path_cd;
 struct pnstruct path_cwd;

   /*************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                   */
   /*************************************************************/
  memset((void*)path name_cd, 0x00, sizeof(struct pnstruct));
  path_cd.qlg_struct.CCSID = 37;
  memcpy(path_cd.qlg_struct.Country_ID,US_const,2);
  memcpy(path_cd.qlg_struct.Language_ID,Language_const,3);
  path_cd.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_cd.qlg_struct.Path_Length = sizeof(mypath_cd)-1;
  path_cd.qlg_struct.Path_Name_Delimiter[0] = '/'
  memcpy(path_cd.pn,mypath_cd,sizeof(mypath_cd)-1);

  if (QlgChdir((Qlg_Path_Name_T *)path name_cd) != 0)
    perror("QlgChdir() error()");
  else
  {
    if (QlgGetcwd(Qlg_Path_Name_T *path_cwd,
                                 sizeof(struct pnstruct)) == NULL)
      perror("QlgGetcwd() error");
    else
      printf("Successful change to new current working directory.");
  }
}
```

Output:

```
Successful change to new current working directory.
```

API introduced: V5R1

QlgGetPathFromFileID()--Get Path Name of Object from Its File ID (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>

Qlg_Path_Name_T *QlgGetPathFromFileID(Qlg_Path_Name_T *buf,
size_t size,Qp0lFID_t fileid);
```

Service Program Name: QP0LLIB2

Default Public Authority: *USE

Threadsafe: Yes

The **QlgGetPathFromFileID()** function, like the **Qp0lGetPathFromFileID()** function, determines an absolute path name of the file identified by *fileid* and stores it in *buf*. The difference is that the **QlgGetPathFromFileID()** function points to a Qlg_Path_Name_T structure, while **Qp0lGetPathFromFileID()** points to a null-terminated character string.

Limited information on the *buf* parameter is provided here. For more information on the *buf* parameter and for a discussion of other parameters, authorities required, return values, and related information, see Qp0lGetPathFromFileID()--Get Path Name of Object from Its File ID.

Parameters

buf

(Output) A pointer to a Qlg_Path_Name_T structure that will be used to hold an absolute path name or a pointer to an absolute path name of the file identified by *fileid*. The path name is not null-terminated within the structure. For more information on the Qlg_Path_Name_T structure, see Path name format.

size

(Input) The number of bytes in the buffer buf.

fileid

(Input) The identifier of the file whose path name is to be returned. This identifier is logged in audit journal entries to identify the file being audited. See the Parent File ID and Object File ID fields of the audit journal entries described in the iSeries Security Reference book.

Related Information

- Qp0lGetPathFromFileID()--Get Path Name of Object from Its File ID

Example

The following example determines the path name of a file, given its file ID. In this example, the *fileid* is hardcoded. More realistically, the *fileid* is obtained from the audit journal entry and passed to **QlgGetPathFromFileID()**.

```
#include <Qp0lstdi.h>
#include <stdio.h>
#include <qtqiconv.h>

void Path_Print(Qlg_Path_Name_T *);

main()
{
  Qp0lFID_t
    fileid = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
              0x00, 0x00, 0x00, 0x00, 0x80, 0xFF, 0xCF, 0x00};

  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[1024]; /* This size must be large enough */
                   /* to contain the path name.      */
  };
  struct pnstruct path;


   /***************************************************/
   /*   Initialize Qlg_Path_Name_T parameters         */
   /***************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);

  if (QlgGetPathFromFileID((Qlg_Path_Name_T *)&path,
                   sizeof(struct pnstruct), fileid) == NULL)
    perror("QlgGetPathFromFileID() error");
  else
  {
    printf("Path retrieved successfully.\n");
    Path_Print((Qlg_Path_Name_T *)&path);
  }
}

void Path_Print(Qlg_Path_Name_T *path_to_print_pointer)
{
 /*************************************************************/
```

```
 /* Print a path name that is in the Qlg_Path_Name_T format.   */
 /**************************************************************/

#define PATH_TYPE_POINTER   0x00000001  /* If flag is on,      */
                         /* input structure contains a pointer */
                         /* to the path name, else the path    */
                         /* name is in contiguous storage      */
                         /* within the qlg structure.          */

  typedef union pn_input_type  /* Format of input path name.   */
  {
    char pn_char_type[256]; /* in contiguous storage */
    char *pn_ptr_type;               /* a pointer              */
  };
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    union pn_input_type   pn;
  };
  struct pnstruct *pns;
  char *path_ptr;

  size_t insz;
  size_t outsz = 1000;
  char outbuf[1000];
  char *outbuf_ptr;
  iconv_t cd;
  size_t ret_iconv;

  /* Indicates to convert from ccsid 13488 to 37.              */
  QtqCode_T toCode   =    {37,0,0,0,0,0};
  QtqCode_T fromCode = {13488,0,0,1,0,0};

  if (path_to_print_pointer != NULL)
  {
    /*********************************************************/
    /* Point to and get the size of the path name.           */
    /*********************************************************/
    pns = (struct pnstruct *)path_to_print_pointer;
    if (path_to_print_pointer->Path_Type & PATH_TYPE_POINTER)
      path_ptr = pns->pn.pn_ptr_type;
    else path_ptr = (char *)(pns->pn.pn_char_type);
    insz = pns->qlg_struct.Path_Length;  /* Get path length.*/

    /*********************************************************/
    /* Initialize the print buffer.                          */
    /*********************************************************/
    outbuf_ptr = (char *)outbuf;
    memset(outbuf_ptr, 0x00, insz);

    /*********************************************************/
    /* Use iconv to convert the CCSID.                       */
    /*********************************************************/
    cd = QtqIconvOpen(&toCode,
                      &fromCode);  /* Open a descriptor*/
    if (cd.return_value == -1)
    { perror("Open conversion descriptor error");
```

```
        return;
      }
      if (0 != ((iconv(cd,
                       (char **)&(path_ptr),
                       &insz,
                       (char **)&(outbuf_ptr),
                       &outsz))))
      {
        ret_iconv= iconv_close(cd);/* Close conversion descriptor*/
        perror("Conversion error");
        return;
      }

      /************************************************************/
      /* Print the name and close the conversion descriptior.     */
      /************************************************************/
      printf("The file's path is: %s\n",outbuf);
      ret_iconv = iconv_close(cd);
    }                             /* path_to_print_pointer != NULL */
  } /* Path_Print */
```

Output:

```
Path retrieved successfully.
The file's path is: /myfile
```

API introduced: V5R1

QlgGetpwnam()--Get User Information for User Name (using NLS-enabled path name)

Syntax

```
#include <pwd.h>

struct qplg_passwd *QlgGetpwnam(const char *name);
```

Service Program Name: QSYPAPI

Default Public Authority: *USE

Threadsafe: No

The **QlgGetpwnam()** function returns a pointer to an object of type struct qplg_passwd containing an entry from the user database with a matching *name*.

Parameters

name

(Input) User profile name.

The struct qplg_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
Qlg_Path_Name_T*	pw_dir	Initial working directory
char *	pw_shell	Initial user program

See getpwnam()--Get User Information for User Name for more on the parameter.

Authorities

*READ authority is required to the user profile associated with the *name*.

Note: Adopted authority is not used.

Return Value

value

QlgGetpwnam was successful. The return value points to static data that is overwritten on each call to this function. This static storage area is also used by the **QlgGetpwuid**() function.

NULL pointer

QlgGetpwnam was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **QlgGetpwnam()** 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.

[EPERM]

The calling job does not have *READ authority to the user profile associated with the *name*.

[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 path name is returned in the default IFS UNICODE CCSID.

Related Information

- The <**pwd.h**> file (see Header Files for UNIX-Type APIs)
- getpwnam()--Get User Information for User Name Qlg getpwnam_r
- getpwnam_r()--Get User Information for User Name
- QlgGetpwnam_r()--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 is 1012. The initial working directory is /home/MYUSER. The initial user program is *LIBL/QCMD.

```
#include <pwd.h>

main()
{
  struct qplg_passwd *pd;

  if (NULL == (pd = QlgGetpwnam("MYUSER")))
     perror("QlgGetpwnam() 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 working directory length is: %d\n",
             pd->pw_dir->Path_Length);
     printf("The initial working directory CCSID is : %d\n",
             pd->pw_dir->CCSID);
     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 working directory length is: 24
  The initial working directory CCSID is : 13488
  The initial user program is: *LIBL/QCMD
```

API introduced: V5R1

QlgGetpwnam_r()--Get User Information for User Name (using NLS-enabled path name)

Syntax

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

int QlgGetpwnam_r(const char *name,
             struct qplg_passwd *pwd,
                        char *buffer,
                       size_t bufsize,
             struct qplg_passwd **result);
```

Service Program Name: QSYPAPI

Default Public Authority: *USE

Threadsafe: Yes

The **QlgGetpwnam_r()** function updates the *qplg_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 qplg_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 qplg_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 qplg_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
Qlg_Path_Name_T*	pw_dir	Initial working directory
char *	pw_shell	Initial user program

See getpwnam_r()--Get User Information for User Name for more on the *pwd*, *result* and other parameters.

Authorities

*READ authority is required to the user profile associated with the *name*.

Return Value

0

QlgGetpwnam_r was successful.

Any other value

Failure: The return value contains an error number indicating the error.

Error Conditions

If **QlgGetpwnam_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.

[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.

[EPERM]

The calling job does not have *READ authority to the user profile associated with the *name*.

[ERANGE]

Insufficient storage was supplied through *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. If there is no message, verify that

the home directory field in the user profile can be displayed.

Usage Notes

The path name is returned in the default IFS UNICODE CCSID.

Related Information

- The <**pwd.h**> file (see Header Files for UNIX-Type APIs)
- getpwnam()--Get User Information for User Name
- 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 is 1012. The initial working 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 qplg_passwd pd;
  qplg_passwd ** tempPwdPtr;
  char pwdbuffer[200];
  int  pwdlinelen = sizeof(pwdbuffer);

  if ((QlgGetpwnam_r("MYUSER",&pd,pwdbuffer,pwdlinelen,tempPwdPtr))!=0    )
     perror("QlgGetpwnam_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 working directory length is: %d\n",
             pd->pw_dir->Path_Length);
     printf("The initial working directory CCSID is : %d\n",
             pd->pw_dir->CCSID);
     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: 0
The intial working directory length is: 24
The intial working directory CCSID is : 13488
The initial user program is: *LIBL/QCMD
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgGetpwuid()--Get User Information for User ID (using NLS-enabled path name)

Syntax

```
#include <pwd.h>

struct qplg_passwd *QlgGetpwuid(uid_t uid);
```

Service Program Name: QSYPAPI

Default Public Authority: *USE

Threadsafe: No

The **QlgGetpwuid()** function returns a pointer to an object of type struct qplg_passwd containing an entry from the user database with a matching user ID (*UID*).

Parameters

UID

(Input) User ID.

The struct qplg_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
Qlg_Path_Name_T*	pw_dir	Initial working directory
char *	pw_shell	Initial user program

See getpwuid()--Get User Information for User ID for more on the parameter.

Authorities

*READ authority is required to the user profile associated with the *UID*.

Note: Adopted authority is not used.

Return Value

value

QlgGetpwuid() was successful. The return value points to static data that is overwritten on each call to this function. This static storage area is also used by the **QlgGetpwnam()** function.

NULL pointer

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

Error Conditions

If **QlgGetpwuid()** 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.

[EPERM]

The calling job does not have *READ authority to the user profile associated with the *UID*.

[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

Th path name is returned in the default IFS UNICODE CCSID

Related Information

- The <**pwd.h**> file (see Header Files for UNIX-Type Functions)
- getpwuid()--Get User Information for User ID
- getpwuid_r()--Get User Information for User ID
- QlgGetpwuid_r()--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 is 1012. The initial working directory is /home/MYUSER. The initial user program is *LIBL/QCMD.

```
#include <pwd.h>

main()
{
  struct qplg_passwd *pd;

  if (NULL == (pd = QlgGetpwuid(22)))
     perror("QlgGetpwuid() 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 working directory length is: %d\n",
             pd->pw_dir->Path_Length);
     printf("The initial working directory CCSID is : %d\n",
             pd->pw_dir->CCSID);
     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 intial working directory length is: 24
  The intial working directory CCSID is : 13488
  The initial user program is: *LIBL/QCMD
```

API introduced: V5R1

QlgGetpwuid_r()--Get User Information for User ID (using NLS-enabled path name)

Syntax

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

int QlgGetpwuid_r(uid_t uid,
     struct qplg_passwd *pwd,
                  char *buffer,
                size_t bufsize,
     struct qplg_passwd **result);
```

Service Program Name: QSYPAPI

Default Public Authority: *USE

Threadsafe: Yes

The **QlgGetpwuid_r()** function updates the *qplg_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) A pointer to a user ID.

pwd

(Input) A pointer to a struct qplg_passwd.

buffer

(Input) A pointer to a buffer from which memory is allocated to hold storage areas referenced by the structure qplg_passwd.

bufsize

(Input) The size of *buffer* in bytes.

result

(Input) A pointer to a location in which a pointer to the updated qplg_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 qplg_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
Qlg_Path_Name_T	pw_dir	Initial working directory
char *	pw_shell	Initial user program

See getpwuid_r()--Get User Information for User ID for more on the *pwd*, *result* and other parameters.

Authorities

*READ authority is required to the user profile associated with the *UID*.

Return Value

0

QlgGetpwuid_r() was successful.

Any other value

Failure: The return value contains an error number indicating the error.

Error Conditions

If **QlgGetpwuid_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.

[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.

[EPERM]

The calling job does not have *READ authority to the user profile associated with the *UID*.

[ERANGE]

Insufficient storage was supplied using *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. If there is no message, verify that the home directory field in the user profile can be displayed.

Usage Notes

The path name is returned in the default IFS UNICODE CCSID.

Related Information

- The <**pwd.h**> file (see Header Files for UNIX-Type Functions)
- getpwuid()--Get User Information for User ID
- 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 is 1012. The intial working 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 qplg_passwd pd;
  passwd ** tempPwdPtr;
  char pwdbuffer[200];
  int  pwdlinelen = sizeof(pwdbuffer);

  if ((QlgGetpwuid_r(22,&pd,pwdbuffer,pwdlinelen,tempPwdPtr))!=0)
     perror("QlgGetpwuid_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 working directory length is: %d\n",
             pd->pw_dir->Path_Length);
     printf("The initial working directory CCSID is : %d\n",
```

```
            pd->pw_dir->CCSID);
    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: 0
  The initial working directory length is: 24
  The initial working directory CCSID is : 13488
  The initial user program is: *LIBL/QCMD
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgLchown()--Change Owner and Group of Symbolic Link (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgLchown(Qlg_Path_Name_T *path, uid_t owner,gid_t
group);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgLchown()** function, like the **lchown()** function, changes the owner and group of a file. The difference is that the **QlgLchown()** function takes a pointer to a Qlg_Path_Name_T structure, while **lchown()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see lchown()--Change Owner and Group of Symbolic Link.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file whose owner and group are being changed. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- lchown()--Change Owner and Group of Symbolic Link
- QlgChmod()--Change File Authorizations
- QlgLstat()--Get File or Link Information
- QlgStat()--Get File Information

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>
#include <Qp0lstdi.h>

main() {

#define mypath_link_name "temp.link"
#define mypath_fn "temp.file"

  const char US_const]3[= "US";
  const char Language_const]4[="ENU";

  struct stat info;
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn]100[; /* This array size must be >= the  */
                           /* length of the path name or this must  */
                           /* be a pointer to the path name.        */
  };
  struct pnstruct path_link;
  struct pnstruct path_fn;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path_link, 0x00, sizeof(struct pnstruct));
  path_link.qlg_struct.CCSID = 37;
  memcpy(path_link.qlg_struct.Country_ID,US_const,2);
  memcpy(path_link.qlg_struct.Language_ID,Language_const,3);
  path_link.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_link.qlg_struct.Path_Length = sizeof(mypath_link_name)-1;
  path_link.qlg_struct.Path_Name_Delimiter]0[ = '/';
  memcpy(path_link.pn,mypath_link_name,sizeof(mypath_link_name)-1);

  memset((void*)&path_fn, 0x00, sizeof(struct pnstruct));
  path_fn.qlg_struct.CCSID = 37;
  memcpy(path_fn.qlg_struct.Country_ID,US_const,2);
  memcpy(path_fn.qlg_struct.Language_ID,Language_const,3);
  path_fn.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_fn.qlg_struct.Path_Length = sizeof(mypath_fn)-1;
  path_fn.qlg_struct.Path_Name_Delimiter]0[ = '/';
  memcpy(path_fn.pn,mypath_fn,sizeof(mypath_fn)-1);

  if (QlgSymlink((Qlg_Path_Name_T *)&path_fn,
                 (Qlg_Path_Name_T *)&path_link) == -1)
    perror("QlgSymlink() error");
  else {
    QlgLstat((Qlg_Path_Name_T *)&path_link, &info);
    printf("original owner was %d and group was %d\n", info.st_uid,
```

```
           info.st_gid);
    if (QlgLchown((Qlg_Path_Name_T *)&path_link, 152, 0) != 0)
      perror("QlgLchown() error");
    else {
      QlgLstat((Qlg_Path_Name_T *)&path_link, &info);
      printf("after QlgLchown(), owner is %d and group is %d\n",
             info.st_uid, info.st_gid);
    }
    QlgUnlink((Qlg_Path_Name_T *)&path_link);
  }
}
```

Output:

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

API introduced: V5R1

QlgLink()--Create Link to File (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgLink(Qlg_Path_Name_T *existing, Qlg_Path_Name_T *new);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgLink()** function, like 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. The difference is that the **QlgLink()** function supports pointers to Qlg_Path_Name_T structures, while **link()** supports pointers to character strings.

Limited information on the *existing* and the *new* parameters is provided here. For more information on these parameters and for a discussion of the authorities required, return values, and related information, see link()--Create Link to File.

Parameters

existing

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of an existing file to which a new link is to be created. For more information on the Qlg_Path_Name_T structure, see Path name format.

new

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name that is the name of the new link. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- link()--Create Link to File
- Qp0lUnlink()--Remove Link to File (using NLS-enabled path name)

Example

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

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <Qp0lstdi.h>

main()
{
  int file_descriptor;
  struct stat info;
#define mypath_fn "link.example.file"
#define mypath_ln "link.example.link"

  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path_fn;
  struct pnstruct path_ln;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path_fn, 0x00, sizeof(struct pnstruct));
  path_fn.qlg_struct.CCSID = 37;
  memcpy(path_fn.qlg_struct.Country_ID,US_const,2);
  memcpy(path_fn.qlg_struct.Language_ID,Language_const,3);
  path_fn.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_fn.qlg_struct.Path_Length = sizeof(mypath_fn)-1;
  path_fn.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_fn.pn,mypath_fn,sizeof(mypath_fn)-1);

  memset((void*)&path_ln, 0x00, sizeof(struct pnstruct));
  path_ln.qlg_struct.CCSID = 37;
  memcpy(path_ln.qlg_struct.Country_ID,US_const,2);
  memcpy(path_ln.qlg_struct.Language_ID,Language_const,3);
  path_ln.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_ln.qlg_struct.Path_Length = sizeof(mypath_ln)-1;
  path_ln.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_ln.pn,mypath_ln,sizeof(mypath_ln)-1);

  if ((file_descriptor = QlgCreat((Qlg_Path_Name_T *)&path_fn, S_IWUSR)) <
0)
    perror("QlgCreat() error");
  else {
    close(file_descriptor);
```

```
    puts("before QlgLink()");
    QlgStat((Qlg_Path_Name_T *)&path_fn,&info);
    printf("   number of links is %hu\n",info.st_nlink);
    if (QlgLink((Qlg_Path_Name_T *)&path_fn,
                (Qlg_Path_Name_T *)&path_ln) != 0) {
      perror("QlgLink() error");
      QlgUnlink((Qlg_Path_Name_T *)&path_fn);
    }
    else {
      puts("after QlgLink()");
      QlgStat((Qlg_Path_Name_T *)&path_fn,&info);
      printf("   number of links is %hu\n",info.st_nlink);
      QlgUnlink((Qlg_Path_Name_T *)&path_ln);
      puts("after first QlgUnlink()");
      QlgLstat((Qlg_Path_Name_T *)&path_fn,&info);
      printf("   number of links is %hu\n",info.st_nlink);
      QlgUnlink((Qlg_Path_Name_T *)&path_fn);
    }
  }
}
```

Output:

```
before QlgLink()
   number of links is 1
after QlgLink()
   number of links is 2
after first QlgUnlink()
   number of links is 1
```

API introduced: V5R1

QlgLstat()--Get File or Link Information (using NLS-enabled path name)

Syntax

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

int QlgLstat(Qlg_Path_Name_T *path,struct stat *buf);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgLstat()** function, like the **lstat()** function, gets status information about a specified file and places it in the area of memory pointed to by *buf*. The difference is that the **QlgLstat()** function takes a pointer to a Qlg_Path_Name_T structure, while **lstat()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see lstat()--Get File or Link Information.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- lstat()--Get File or Link Information
- QlgChmod()--Change File Authorizations (using NLS-enabled path name)
- QlgChown()--Change Owner and Group of File (using NLS-enabled path name)
- QlgCreat()--Create or Rewrite File (using NLS-enabled path name)
- QlgLink()--Create Link to File (using NLS-enabled path name)
- QlgMkdir()--Make Directory (using NLS-enabled path name)
- QlgReadlink()--Read Value of Symbolic Link (using NLS-enabled path name)
- QlgStat()--Get File Information (using NLS-enabled path name)
- QlgSymlink()--Make Symbolic Link (using NLS-enabled path name)
- QlgUtime()--Set File Access and Modification Times (using NLS-enabled path name)
- Qp0lUnlink()--Remove Link to File (using NLS-enabled path name)

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>
#include <Qp0lstdi.h>

main() {

  struct stat info;
  int file_descriptor;
 #define mypath_fn "temp.file"
 #define mypath_ln "temp.link"

  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or this must */
                  /* be a pointer to the path name.       */
  };
 struct pnstruct path_fn;
 struct pnstruct path_ln;

   /*************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                   */
   /*************************************************************/
  memset((void*)&path_fn, 0x00, sizeof(struct pnstruct));
  path_fn.qlg_struct.CCSID = 37;
  memcpy(path_fn.qlg_struct.Country_ID,US_const,2);
  memcpy(path_fn.qlg_struct.Language_ID,Language_const,3);
  path_fn.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_fn.qlg_struct.Path_Length = sizeof(mypath_fn)-1;
  path_fn.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_fn.pn,mypath_fn,sizeof(mypath_fn)-1);

  memset((void*)&path_ln, 0x00, sizeof(struct pnstruct));
  path_ln.qlg_struct.CCSID = 37;
  memcpy(path_ln.qlg_struct.Country_ID,US_const,2);
  memcpy(path_ln.qlg_struct.Language_ID,Language_const,3);
  path_ln.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_ln.qlg_struct.Path_Length = sizeof(mypath_ln)-1;
  path_ln.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_ln.pn,mypath_ln,sizeof(mypath_ln)-1);

 if ((file_descriptor = QlgCreat((Qlg_Path_Name_T *)&path_fn, S_IWUSR)) < 0)
    perror("QlgCreat() error");
  else {
    close(file_descriptor);
    if (QlgLink((Qlg_Path_Name_T *)&path_fn,
```

```
                (Qlg_Path_Name_T *)&path_ln)
                                    !=0
      perror("QlgLink() error");
    else {
      if (QlgLstat((Qlg_Path_Name_T *)&path_ln, &info) != 0)
        perror("QlgLstat() error");
      else {
        puts("QlgLstat() 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);
      }
      QlgUnlink((Qlg_Path_Name_T *)&path_ln);
    }
    QlgUnlink((Qlg_Path_Name_T *)&path_fn);
  }
}
```

Output:

```
QlgLstat() returned:
  inode:   8477
 dev id:   0
   mode:   00008080
  links:   2
    uid:   1782
    gid:   0
```

QlgLstat64()--Get File or Link Information (large file enabled and using NLS-enabled path name)

Syntax

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

int QlgLstat64(Qlg_Path_Name_T *path, struct stat64 *buf);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgLstat64()** function, like the **lstat64()** function, gets status information about a specified file and places it in the area of memory pointed to by *buf*. The difference is that the **QlgLstat64()** function takes a pointer to a Qlg_Path_Name_T structure, while **lstat64()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see lstat64()--Get File or Link Information (Large File Enabled) or lstat()--Get File or Link Information.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- lstat64()--Get File or Link Information (large file enabled and using NLS-enabled path name)
- lstat()--Get File or Link Information (using NLS-enabled path name)
- QlgChmod()--Change File Authorizations (using NLS-enabled path name)
- QlgChown()--Change Owner and Group of File (using NLS-enabled path name)
- QlgCreat()--Create or Rewrite File (using NLS-enabled path name)
- QlgLink()--Create Link to File (using NLS-enabled path name)
- QlgMkdir()--Make Directory (using NLS-enabled path name)
- QlgReadlink()--Read Value of Symbolic Link (using NLS-enabled path name)
- QlgStat()--Get File Information (using NLS-enabled path name)
- QlgSymlink()--Make Symbolic Link (using NLS-enabled path name)
- QlgUtime()--Set File Access and Modification Times (using NLS-enabled path name)
- Qp0lUnlink()--Remove Link to File (using NLS-enabled path name)

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 <Qp0lstdi.h>

main() {
  struct stat64 info;
  int file_descriptor;
#define mypath_fn "temp.file"
#define mypath_ln "temp.link"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path_fn;
  struct pnstruct path_ln;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path_fn, 0x00, sizeof(struct pnstruct));
  path_fn.qlg_struct.CCSID = 37;
  memcpy(path_fn.qlg_struct.Country_ID,US_const,2);
  memcpy(path_fn.qlg_struct.Language_ID,Language_const,3);
  path_fn.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_fn.qlg_struct.Path_Length = sizeof(mypath_fn)-1;
  path_fn.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_fn.pn,mypath_fn,sizeof(mypath_fn)-);

  memset((void*)&path_ln, 0x00, sizeof(struct pnstruct));
  path_ln.qlg_struct.CCSID = 37;
  memcpy(path_ln.qlg_struct.Country_ID,US_const,2);
  memcpy(path_ln.qlg_struct.Language_ID,Language_const,3);
  path_ln.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_ln.qlg_struct.Path_Length = sizeof(mypath_ln)-1;
  path_ln.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_ln.pn,mypath_ln,sizeof(mypath_ln)-);

  if ((file_descriptor = QlgCreat64((Qlg_Path_Name_T *)&path_fn, S_IWUSR)) <
    perror("QlgCreat64() error");
  else {
    close(file_descriptor);
    if (QlgLink((Qlg_Path_Name_T *)&path_fn,
```

```
                (Qlg_Path_Name_T *)&path_ln) != 0)
      perror("QlgLink() error");
    else {
      if (QlgLstat64((Qlg_Path_Name_T *)&path_ln, &info) != 0)
        perror("QlgLstat64() error");
      else {
        puts("QlgLstat64() 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);
      }
      QlgUnlink((Qlg_Path_Name_T *)&path_ln);
    }
    QlgUnlink((Qlg_Path_Name_T *)&path_fn);
  }
}
```

Output:

```
QlgLstat() returned:
  inode:   258
 dev id:   1
   mode:   00008080
  links:   2
    uid:   137
    gid:   500
   size:   18
```

QlgMkdir()--Make Directory (using NLS-enabled path name)

Syntax

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

int QlgMkdir(Qlg_Path_Name_T *path, mode_t mode);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgMkdir()** function, like the **mkdir()** function, creates a new, empty directory whose name is defined by *path*. The difference is that the **QlgMkdir()** function takes a pointer to a Qlg_Path_Name_T structure, while **mkdir()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see mkdir()--Make Directory.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the directory to be created. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- mkdir()--Make Directory
- QlgChmod()--Change File Authorizations (using NLS-enabled path name)
- QlgStat()--Get File Information (using NLS-enabled path name)
- QlgPathconf()--Get Configurable Path Name Variables (using NLS-enabled path name)

Example

The following example creates a new directory:

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

```
#include <unistd.h>
#include <stdio.h>
main() {

 #define mypath "new_dir"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  const char mypath_DOT_DOT[3] = "..";

  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or this must */
                  /* be a pointer to the path name.       */
  };
  struct pnstruct path;
  struct pnstruct path_DOT_DOT;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);


  memset((void*)&path_DOT_DOT, 0x00, sizeof(struct pnstruct));
  path_DOT_DOT.qlg_struct.CCSID = 37;
  memcpy(path_DOT_DOT.qlg_struct.Country_ID,US_const,2);
  memcpy(path_DOT_DOT.qlg_struct.Language_ID,Language_const,3);
  path_DOT_DOT.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_DOT_DOT.qlg_struct.Path_Length = sizeof(mypath_DOT_DOT)-1;
  path_DOT_DOT.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_DOT_DOT.pn,mypath_DOT_DOT,sizeof(mypath_DOT_DOT)-1);

  if (QlgMkdir((Qlg_Path_Name_T *)&path,
               S_IRWXU|S_IRGRP|S_IXGRP) != 0)
    perror("QlgMkdir() error");
  else if (QlgChdir((Qlg_Path_Name_T *)&path) != 0)
    perror("first QlgChdir() error");
  else if (QlgChdir((Qlg_Path_Name_T *)&path_DOT_DOT) != 0)
    perror("second QlgChdir() error");
  else if (QlgRmdir((Qlg_Path_Name_T *)&path) != 0)
    perror("QlgRmdir() error");
  else
    puts("success!");
}
```

API introduced: V5R1

QlgMkfifo()--Make FIFO Special File (using NLS-enabled path name)

Syntax

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

int QlgMkfifo(const Qlg_Path_Name_T *path,
              mode_t mode);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgMkfifo()** function, like the **mkfifo()** function, creates a new FIFO special file whose name is defined by *path*. The difference is that the **QlgMkfifo()** function takes a pointer to a Qlg_Path_Name_T structure, while **mkfifo()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see mkfifo()--Make FIFO Special File.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the FIFO to be created. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- mkfifo()--Make FIFO Special File
- QlgChmod()--Change File Authorizations (using NLS-enabled path name)
- QlgStat()--Get File Information (using NLS-enabled path name)

Example

The following example creates a new FIFO:

```
#include <sys/stat.h>
#include <stdio.h>
#include <string.h>
#include <Qlg.h>

void main()
{
 typedef struct pnstruct
    {
     Qlg_Path_Name_T qlg_struct;
     char[100] pn;  /* This size must be >= the path */
                              /* name length or a pointer to   */
                              /* the path name.                */
    };
 struct pnstruct path;

 char *mypath = "/newFIFO";

 /***************************************************************/
 /*   Initialize Qlg_Path_Name_T structure.                     */
 /***************************************************************/
 memset((void*)path name, 0x00, sizeof(struct pnstruct));
 path.qlg_struct.CCSID = 37;
 memcpy(path.qlg_struct.Country_ID, "US", 2);
 memcpy(path.qlg_struct.Language_ID, "ENU", 3);
 path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
 path.qlg_struct.Path_Length = strlen(mypath);
 path.qlg_struct.Path_Name_Delimiter = '/';
 memcpy(path.pn, mypath, strlen(mypath));

 if (QlgMkfifo((Qlg_Path_Name_T *)path name,
              S_IRWXU|S_IRWXO) != 0)
   perror("QlgMkfifo() error");
 else
   puts("success!");

 return;
}
```

QlgOpen()--Open a File (using NLS-enabled path name)

Syntax

```
#include <fcntl.h>
#include <stdio.h>
#include <Qp0lstdi.h>

int QlgOpen(Qlg_Path_Name_T *Path_Name,
int oflag, . . .);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes for open() API.

The **QlgOpen**() function, like the **open()** function, opens a file or creates a new, empty file whose name is defined by *path* and returns a number called a **file descriptor**. The difference is that the **QlgOpen()** function takes a pointer to a Qlg_Path_Name_T structure, while **open()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, usage notes, return values, and related information, see open()--Open a File.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file to be opened. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- open()--Open a File
- QlgCreat()--Create or Rewrite File (using NLS-enabled path name)
- QlgOpen64()--Open File (large file enabled and using NLS-enabled path name)
- QlgStat()>--Get File Information (using NLS-enabled path name)

Example

The following example creates and opens an output file for exclusive access. This program was stored in a source file with CCSID 37, so the constant string "newfile" will be compiled in CCSID 37. Therefore, the language and country or region specified are United States English, and the CCSID specified is 37.

```
#include <fcntl.h>
#include <stdio.h>
#include <Qp0lstdi.h>

main()
{
  int fildes;

  const char US_const[3]= "US";
  const char Language_const[4]="ENU";

  struct pnstruct
    {
     Qlg_Path_Name_T  qlg_struct;
     char             pn[7];
    };
  struct pnstruct pns;
  struct pnstruct *pns_ptr = NULL;

  char fn[]="newfile";

  memset((void*)&pns, 0x00, sizeof(struct pnstruct));
  pns.qlg_struct.CCSID = 37;
  memcpy(pns.qlg_struct.Country_ID,US_const,2);
  memcpy(pns.qlg_struct.Language_ID,Language_const,3);;
  pns.qlg_struct.Path_Type = 0;
  pns.qlg_struct.Path_Length = sizeof(fn) - 1;
  pns.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(pns.pn,fn,sizeof(fn)-1);

  pns_ptr = &pns;
  if(fildes = QlgOpen((Qlg_Path_Name_T *)pns_ptr,
           O_WRONLY|O_CREAT|O_EXCL, S_IRWXU)) == -1)
  {
        perror("QlgOpen() error");
  }

}
```

QlgOpen64()--Open File (large file enabled and using NLS-enabled path name)

Syntax

```
#include <fcntl.h>

int QlgOpen64(Qlg_Path_Name_T *path, int oflag, . . .);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgOpen64()** function, like the **open64()** and **open()** functions, opens a file and returns a number called a file descriptor. **QlgOpen64()** differs from **open64()** in that the **open64()** function takes a pointer to a Qlg_Path_Name_T structure, while **open64()** takes a pointer to a character string. **QlgOpen64()** differs from **open()** in that it automatically opens a file with the O_LARGEFILE flag set.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see open()--Open a File or QlgOpen64()--Open File (Large File Enabled).

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file to be opened. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- open()--Open a File
- QlgCreat()--Create or Rewrite File (using NLS-enabled path name)
- QlgStat()--Get File Information (using NLS-enabled path name)

API introduced: V5R1

QlgOpendir()--Open Directory (using NLS-enabled path name)

Syntax

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

DIR *QlgOpendir(Qlg_Path_Name_T *dirname);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgOpendir**() function, like the **opendir**() function, opens a directory so it can be read. The difference is that the **QlgOpendir()** function takes a pointer to a Qlg_Path_Name_T structure, while the **opendir()** function takes a pointer to a character string. The **QlgOpendir()** function opens a directory so it can be read with the **QlgReaddir**() function.

Names returned on calls to **QlgReaddir()** are returned in the coded character set identifier (CCSID) specified at the time the directory is opened. **QlgOpendir()** allows the CCSID to be specified in the Qlg_Path_Name_T structure. **opendir**() uses the CCSID that is in effect for the current job at the time the **opendir()** function is called. See opendir()--Open Directory for more on the job CCSID.

Limited information on the *dirname* parameter is provided here. For more information on the *dirname* parameter and for a discussion of authorities required, return values, and related information, see opendir()--Open Directory.

Parameters

dirname

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the directory to be opened. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- opendir()--Open Directory
- QlgReaddir()--Read Directory Entry (using NLS-enabled path name)
- QlgSpawn()--Spawn Process (using NLS-enabled path name)
- QlgSpawnp()--Spawn Process with Path (using NLS-enabled fileh name)

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;
  int count;
  struct stat info;

  typedef struct my_dirent_lg
  {
    struct dirent_lg *entry;
    char          d_lg_name[1];
  };

  struct my_dirent_lg lg_struct;
  struct dirent_lg *entry;

  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[1025]; /* This array size must be >=   */
                            /* the length of the path name or  */
                            /* this must be a pointer to the   */
                            /* path name.                      */
  };

  struct pnstruct path;
  struct pnstruct path_to_stat;
  char *temp_char_path[1025];

   /*************************************************************/
   /*   Initialize Qlg_Path_Name_T structure, since the path name */
   /*   was not in the Qlg_Path_Name_T format when this function  */
   /*   was called.                                               */
   /*************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  path.qlg_struct.Path_Length = strlen(fn);
  memcpy(path.pn,fn,strlen(fn));

  for (count=0; count < indent; count++) printf("  ");
  printf("%s\n", fn);

  if ((dir = QlgOpendir((Qlg_Path_Name_T *)&path)) == NULL)
```

```
    perror("QlgOpendir() error");
  else
  {
    path_to_stat = path;

    while ((entry = QlgReaddir(dir)) != NULL)
    {

      if
        (entry->d_lg_name[0] != '.')
      {
        /* Concat the components of the path name into a  */
        /* Qlg_Path_Name_T structure that is used on the  */
        /* next function that is called. Clear and        */
        /* use a temporary buffer to ensure that only     */
        /* characters returned by QlgReaddir() are        */
        /* included in the concatenated path name         */
        /* structure.                                     */
        strcpy(path_to_stat.pn,path.pn);
        strcat(path_to_stat.pn, "/");
        memset(temp_char_path, 0x00,1025);
        memcpy(temp_char_path,
               entry->d_lg_name,entry->d_lg_qlg.Path_Length);

        strcat(path_to_stat.pn,(char *)&temp_char_path);

        /* Calculate the size of the path, including the  */
        /* length of the path specified on the open, the  */
        /* length of the name returned by QlgReaddir(),   */
        /* and the delimiter.                             */

        path_to_stat.qlg_struct.Path_Length =
          (path.qlg_struct.Path_Length +
           entry->d_lg_qlg.Path_Length + 1);


        /* Call QlgStat() to determine if the path name   */
        /* is a directory.                                */
        if (QlgStat((Qlg_Path_Name_T *)&path_to_stat,
                    &info) != 0)
        {
          fprintf(stderr, "QlgStat() error on %s: %s\n",
                  path_to_stat.pn,
                  strerror(errno));
        }
        else if (S_ISDIR(info.st_mode))
        {
          /* this a directory so loop to open its objects.*/
          traverse(path_to_stat.pn, indent+1);
        }
        else printf(" %s\n",path_to_stat.pn);
      }
    }
    closedir(dir);
  }
}


main() {
```

```
  puts("Directory structure:");
  traverse("/etc", 0);
}
```

Output:

```
Directory structure:
/etc
  /etc/samples
    /etc/samples/IBM
  /etc/IBM
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgPathconf()--Get Configurable Path Name Variables (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

long QlgPathconf(Qlg_Path_Name_T *path, int name);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgPathconf()** function, like the **pathconf()** function, lets an application determine the value of a configuration variable (*name*) associated with a particular file or directory (*path*). The difference is that the **QlgPathconf()** function takes a pointer to a Qlg_Path_Name_T structure, while **pathconf()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see pathconf()--Get Configurable Path Name Variables.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name for which the value of the configuration variable is requested. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- fpathconf()--Get Configurable Path Name Variables by Descriptor
- pathconf()--Get Configurable Path Name Variables
- QlgChown()--Change Owner and Group of File (using NLS-enabled path name)

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;
#define mypath "/"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                      */
   /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  errno = 0;
  puts("examining NAME_MAX limit for root filesystem");
  if ((result = QlgPathconf((Qlg_Path_Name_T *)&path,
                                       _PC_NAME_MAX)) == -1)
    if (errno == 0)
      puts("There is no limit to NAME_MAX.");
    else perror("QlgPathconf() error");
  else
    printf("NAME_MAX is %ld\n", result);
}
```

Output:

```
examining NAME_MAX limit for root filesystem
NAME_MAX is 255
```

API introduced: V5R1

QlgProcessSubtree()--Process a Path Name (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>

int QlgProcessSubtree (
 Qlg_Path_Name_T           *Path_Name,
 uint                       Subtree_level,
 Qp0l_Objtypes_List_t      *Objtypes_array_ptr,
 uint                       Local_remote_obj,
 Qp0l_IN_EXclusion_List_t *IN_EXclusion_ptr,
 uint                       Err_recovery_action,
 Qp0l_User_Function_t      *UserFunction_ptr,
 void                      *Function_CtlBlk_ptr, ...);
```

Service Program Name: QP0LLIB2

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

For a description of this function and information on its parameters, authorities required, return values, error conditions, error messages, usage notes, and related information, see Qp0lProcessSubtree()--Process a Path Name.

API introduced: V5R1

QlgReaddir()--Read Directory Entry (using NLS-enabled path name)

Syntax

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

struct dirent_lg *QlgReaddir(DIR *dirp);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: No; see Usage Notes.

The **QlgReaddir()** function, like the **readdir()** function, returns a pointer to a structure describing the next directory entry in the directory stream associated with *dirp*. The difference is that the **QlgReaddir()** function takes a pointer to a dirent_lg structure, while **readdir()** takes a pointer to a dirent structure.

Limited information on the *dirp* parameter is provided here. For more information on the *dirp* parameter and for a discussion of authorities required, return values, and related information, see readdir()--Read Directory Entry.

Parameters

dirp

(Input) A pointer to DIR that refers to the open directory stream to be read. This pointer is returned by **QlgOpendir()**.

A dirent_lg structure has the following contents:

char	d_reserved1[16]	Reserved.
unsigned int	d_fileno_gen_id	The generation ID associated with the file ID.
ino_t	d_fileno	The file ID of the file. This number uniquely identifies the object within a file system.
unsigned int	d_reclen	The length of the directory entry in bytes.
int	d_reserved3	Reserved.
char	d_reserved4[6]	Reserved.
char	d_reserved5[2]	Reserved.

Qlg_Path_Name_T	d_lg_name	A Qlg_Path_Name_T that gives the name of a file in the directory. The path name is not null-terminated within the structure. The structure also provides National Language Support information, which includes ccsid, country_id, and language_id. This structure has a maximum length of {_QP0L_DIR_NAME_LG} bytes. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- readdir()--Read Directory Entry
- QlgOpendir()--Open Directory (using NLS-enabled path name)
- QlgPathconf()--Get Configurable Path Name Variables (using NLS-enabled path name)

Example

The following example reads the contents of a root directory:

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

main() {

  typedef struct my_dirent_lg
  {
    struct dirent_lg *entry;
    char             d_lg_name[1];
  };

  struct my_dirent_lg lg_struct;
  struct dirent_lg *entry;
#define mypath "/"
  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100];     /* This array size must be >= */
                                /* the length of the path name    */
                                /* or this must be a pointer      */
                                /* to the path name.              */
  };
```

```
  struct pnstruct path;
  DIR     *dir;

   /**************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                    */
   /**************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if ((dir = QlgOpendir((Qlg_Path_Name_T *)&path)) == NULL)
    perror("QlgOpendir() error");
  else {
    puts("contents of root:");
    while ((entry = QlgReaddir(dir)) != NULL)
      printf("  %s\n", entry->d_lg_name);
    closedir(dir);
  }
}
```

Output:

```
contents of root:
  .
  ..
  QSYS.LIB
  QDLS
  QOpenSys
  QOPT
  home
```

API introduced: V5R1

QlgReaddir_r()--Read Directory Entry (using NLS-enabled path name)

Syntax

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

int QlgReaddir_r(DIR *dirp, struct dirent_lg *entry,
                    struct dirent_lg **result);
```

Service Program Name: QP0LLIBTS

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgReaddir_r()** function, like the **readdir_r()** function, initializes a structure that is referenced by *entry* to represent the next directory entry in the directory stream that is associated with *dirp*. The difference is that the **QlgReaddir_r()** *dirp* parameter points to a dirent_lg structure, while the **readdir_r()** *dirp* parameter points to a dirent structure.

The QlgReaddir_r functions stores a pointer to the *entry* structure at the location referenced by *result*.

Limited information on the *dirp* parameter, the *entry* parameter, and the *result* parameter is provided here. For more information on these parameters and for a discussion of authorities required, return values, and related information, see readdir_r()--Read Directory Entry.

Parameters

dirp

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

entry

(Output) A pointer to a dirent_lg structure in which the directory entry is to be placed.

result

(Output) A pointer to a pointer to a dirent_lg structure. Upon successfully reading a directory entry, this dirent_lg pointer is set to the same value as *entry*. Upon reaching the end of the directory stream, this pointer is set to NULL.

A dirent_lg structure has the following contents:

char	d_reserved1[16]	Reserved.
unsigned int	d_fileno_gen_id	The generation ID associated with the file ID.
ino_t	d_fileno	The file ID of the file. This number uniquely identifies the object within a file system.

unsigned int	d_reclen	The length of the directory entry in bytes.
int	d_reserved3	Reserved.
char	d_reserved4[6]	Reserved.
char	d_reserved5[2]	Reserved.
Qlg_Path_Name_T	d_lg_name	A Qlg_Path_Name_T structure that gives the name of a file in the directory. The path name is not null-terminated within the structure. The structure also provides National Language Support information, which includes ccsid, country_id, and language_id. This structure has a maximum length of {_QP0L_DIR_NAME_LG} bytes. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- readdir()--Read Directory Entry
- QlgOpendir()--Open Directory (using NLS-enabled path name)
- QlgPathconf()--Get Configurable Path Name Variables (using NLS-enabled path name)

Example

The following example reads the contents of a root directory:

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

main() {
  int return_code;
  DIR *dir;
  struct dirent_lg entry;
  struct dirent_lg *result;

  typedef struct my_dirent_lg
  {
    struct dirent_lg *entry;
    char             d_lg_name[1];
  };
  struct my_dirent_lg lg_struct;

#define mypath "/"
  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  typedef struct pnstruct
```

```
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100];  /* This array size must be >=    */
                         /* the length of the path name or this */
                         /* must be a pointer to the path name. */
  };
  struct pnstruct path;

   /*****************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                       */
   /*****************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if ((dir = QlgOpendir((Qlg_Path_Name_T *)&path)) == NULL)
    perror("QlgOpendir() error");
  else {
    puts("contents of root:");
    for (return_code = QlgReaddir_r(dir, &entry, &result);
         result != NULL && return_code == 0;
         return_code = QlgReaddir_r(dir, &entry, &result))
      printf("  %s\n", entry.d_lg_name);
    if (return_code != 0)
      perror("QlgReaddir_r() error");
    closedir(dir);
  }
}
```

Output:

```
contents of root:
  .
  ..
  QSYS.LIB
  QDLS
  QOpenSys
  QOPT
  home
```

API introduced: V5R1

QlgReadlink()--Read Value of Symbolic Link (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgReadlink(Qlg_Path_Name_T *path, Qlg_Path_Name_T *buf,
size_t bufsiz);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgReadlink()** function, like the **readlink()** function, places the contents of the symboliclink *path* in the buffer *buf*. The difference is that the **QlgReadlink()** function uses pointers to Qlg_Path_Name_T structures, while **readlink()** uses pointers to character strings.

Limited information on the *path* parameter, the *buf* parameter, and the *size* parameter is provided here. For more information on these parameters and for a discussion authorities required, return values, and related information, see readlink()--Read Value of Symbolic Link.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the symbolic link. For more information on the Qlg_Path_Name_T structure, see Path name format.

buf

(Output) A pointer to the area in which the contents of the link should be stored. For more information on the Qlg_Path_Name_T structure, see Path name format.

bufsiz

(Input) The size of *buf* in bytes.

Related Information

- readlink()--Read Value of Symbolic Link
- QlgLstat()--Get File or Link Information (using NLS-enabled path name)
- QlgStat()--Get File Information (using NLS-enabled path name)
- QlgSymlink()--Make Symbolic Link (using NLS-enabled path name)

- Qp0lUnlink()--Remove Link to File

Example

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

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <Qp0lstdi.h>

main() {
  int file_descriptor;

 #define mypath_fn "readlink.file"
 #define mypath_sl "readlink.symlink"

 const char US_const[3]= "US";
 const char Language_const[4]="ENU";
 typedef struct pnstruct
    {
     Qlg_Path_Name_T qlg_struct;
     char pn[100]; /* This array size must be >= the length */
                         /* of the path name or this must be a    */
                         /* pointer to the path name.             */

    };

 struct pnstruct path_fn;
 struct pnstruct path_sl;
 struct pnstruct path_buf;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)path name_fn, 0x00, sizeof(struct pnstruct));
  path_fn.qlg_struct.CCSID = 37;
  memcpy(path_fn.qlg_struct.Country_ID,US_const,2);
  memcpy(path_fn.qlg_struct.Language_ID,Language_const,3);
  path_fn.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_fn.qlg_struct.Path_Length = sizeof(mypath_fn)-1;
  path_fn.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_fn.pn,mypath_fn,sizeof(mypath_fn)-1);

  memset((void*)path name_sl, 0x00, sizeof(struct pnstruct));
  path_sl.qlg_struct.CCSID = 37;
  memcpy(path_sl.qlg_struct.Country_ID,US_const,2);
  memcpy(path_sl.qlg_struct.Language_ID,Language_const,3);
  path_sl.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_sl.qlg_struct.Path_Length = sizeof(mypath_sl)-1;
  path_sl.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_sl.pn,mypath_sl,sizeof(mypath_sl)-1);
```

```
  if ((file_descriptor = QlgCreat((Qlg_Path_Name_T *)path name_fn, S_IWUSR))
< 0)
    perror("QlgCreat() error");
  else {
    close(file_descriptor);
    if (QlgSymlink((Qlg_Path_Name_T *)path name_fn,
                   (Qlg_Path_Name_T *)path name_sl) != 0)
      perror("QlgSymlink() error");
    else {
      if (QlgReadlink((Qlg_Path_Name_T *)path name_sl,
                      (Qlg_Path_Name_T *)path name_buf,
                      sizeof(path_buf)) < 0)
        perror("QlgReadlink() error");
      else printf("QlgReadlink() returned '%s' for '%s'\n",
                                 path name_buf.pn,
                                 path name_sl.pn);

      QlgUnlink((Qlg_Path_Name_T *)path name_sl);
    }
    QlgUnlink((Qlg_Path_Name_T *)path name_fn);
  }
}
```

Output:

```
QlgReadlink() returned 'readlink.file' for 'readlink.symlink'
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgRenameKeep()--Rename File or Directory, Keep "new" If It Exists (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>

int QlgRenameKeep(Qlg_Path_Name_T *old, Qlg_Path_Name_T *new);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgRenameKeep()** function, like the **Qp0lRenameKeep()** function, renames a file or a directory specified by *old* to the name given by *new*. The difference is that the **QlgRenameKeep()** function takes pointers to Qlg_Path_Name_T structures, while **Qp0lRenameKeep()** takes pointers to character strings.

Limited information on the *old* and *new* parameters is provided here. For more information on these parameters and for a discussion of the authorities required, return values, and related information, see Qp0lRenameKeep()--Rename File or Directory, Keep "new" If It Exists.

Parameters

old

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to the path name of the file to be renamed. For more information on the Qlg_Path_Name_T structure, see Path name format.

new

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to the path name of the new name for the file. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- Qp0lRenameKeep()--Rename File or Directory, Keep "new" If It Exists
- QlgPathconf()--Get Configurable Path Name Variables (using NLS-enabled path name)
- QlgRenameUnlink()--Rename File or Directory, Unlink "new" If It Exists (using NLS-enabled path name)

Example

When you pass two file names to this example, it changes the first file name to the second file name using **QlgRenameKeep()**.

```
#include <Qp0lstdi.h>
#include <stdio.h>

int main(int argc, char **argv)
{

  if ( argc != 3 )
  {
    printf( "Usage: %s old_fn new_fn\n", argv[0]);
    perror ( "Could not rename file" );
  }

  else
  {
    const char US_const[3]= "US";
    const char Language_const[4]="ENU";
    typedef struct pnstruct
    {
      Qlg_Path_Name_T qlg_struct;
                                  /*** EXTRA STORAGE MAY BE NEEDED ***/
      char pn[1025];  /* This size must be >= the path */
                                  /* name length or a pointer to   */
                                  /* the path name.                */
    };
    struct pnstruct path_old;
    struct pnstruct path_new;

    struct pnstruct *path_old_ptr;
    struct pnstruct *path_new_ptr;


    memset((void*)&path_old, 0x00, sizeof(struct pnstruct));
    path_old_ptr = &path_old;

    path_old.qlg_struct.CCSID = 37;
    memcpy(path_old.qlg_struct.Country_ID,US_const,2);
    memcpy(path_old.qlg_struct.Language_ID,Language_const,3);;
    path_old.qlg_struct.Path_Type = 0;
    path_old.qlg_struct.Path_Length = strlen(argv[1]);
    path_old.qlg_struct.Path_Name_Delimiter[0] = '/';
    memcpy(path_old.pn,argv[1],sizeof(argv[1])-1);


    memset((void*)&path_new, 0x00, sizeof(struct pnstruct));
    path_new_ptr = &path_new;

    path_new.qlg_struct.CCSID = 37;
    memcpy(path_new.qlg_struct.Country_ID,US_const,2);
```

```
    memcpy(path_new.qlg_struct.Language_ID,Language_const,3);;
    path_new.qlg_struct.Path_Type = 0;
    path_new.qlg_struct.Path_Length = strlen(argv[2]);
    path_new.qlg_struct.Path_Name_Delimiter[0] = '/';
    memcpy(path_new.pn,argv[2],sizeof(argv[2])-1);


    if (QlgRenameKeep((Qlg_Path_Name_T *)path_old_ptr,
                      (Qlg_Path_Name_T *)path_new_ptr ) != 0)
    {perror ( "Could not rename file." ); }
    else {perror ( "File renamed." ); }
  }

}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgRenameUnlink()--Rename File or Directory, Unlink "new" If It Exists (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>

int QlgRenameUnlink(Qlg_Path_Name_T *old, Qlg_Path_Name_T *new);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgRenameUnlink()** function, like the **Qp0lRenameUnlink()** function, renames a file or a directory specified by *old* to the name given by *new*. The difference is that the **QlgRenameUnlink()** function takes a pointer to a Qlg_Path_Name_T structure, while **Qp0lRenameUnlink()** takes a pointer to a character string.

Limited information on the *old* and *old* parameters is provided here. For more information on these parameters and for a discussion of the authorities required, return values, and related information, see Qp0lRenameUnlink()--Rename File or Directory, Unlink "new" If It Exists.

Parameters

old

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file to be renamed. For more information on the Qlg_Path_Name_T structure, see Path name format.

new

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the new name of the file. For more information on the Qlg_Path_Name_T structure, Path name format.

Related Information

- Qp0lRenameUnlink()--Rename File or Directory, Unlink "new" If It Exists
- QlgPathconf()--Get Configurable Path Name Variables (using NLS-enabled path name)
- QlgRenameKeep()--Rename File or Directory, Keep "new" If It Exists (using NLS-enabled path

name)

Example

When you pass two file names to this example, it tries to change the file name from the first name to the second using **QlgRenameUnlink()**.

```
#include <Qp0lstdi.h>
#include <stdio.h>

int main(int argc, char **argv)
{

  if ( argc != 3 )
  {
    printf( "Usage: %s old_fn new_fn\n", argv[0]);
    perror ( "Could not unlink the file" );
  }

  else
  {
    const char US_const[3]= "US";
    const char Language_const[4]="ENU";
    typedef struct pnstruct
    {
      Qlg_Path_Name_T qlg_struct;
                                  /*** EXTRA STORAGE MAY BE NEEDED ***/
      char pn[1025];               /* This size must be >= the path */
                                  /* name length or a pointer to   */
                                  /* the path name.                */
    };
    struct pnstruct path_old;
    struct pnstruct path_new;

    struct pnstruct *path_old_ptr;
    struct pnstruct *path_new_ptr;


    memset((void*)&path_old, 0x00, sizeof(struct pnstruct));
    path_old_ptr = &path_old;

    path_old.qlg_struct.CCSID = 37;
    memcpy(path_old.qlg_struct.Country_ID,US_const,2);
    memcpy(path_old.qlg_struct.Language_ID,Language_const,3);;
    path_old.qlg_struct.Path_Type = 0;
    path_old.qlg_struct.Path_Length = strlen(argv[1]);
    path_old.qlg_struct.Path_Name_Delimiter[0] = '/';
    memcpy(path_old.pn,argv[1],sizeof(argv[1]));


    memset((void*)&path_new, 0x00, sizeof(struct pnstruct));
    path_new_ptr = &path_new;

    path_new.qlg_struct.CCSID = 37;
```

```
    memcpy(path_new.qlg_struct.Country_ID,US_const,2);
    memcpy(path_new.qlg_struct.Language_ID,Language_const,3);;
    path_new.qlg_struct.Path_Type = 0;
    path_new.qlg_struct.Path_Length = strlen(argv[2]);
    path_new.qlg_struct.Path_Name_Delimiter[0] = '/';
    memcpy(path_new.pn,argv[2],sizeof(argv[2]));


    if (QlgRenameUnlink((Qlg_Path_Name_T *)path_old_ptr,
                        (Qlg_Path_Name_T *)path_new_ptr ) != 0)
    {perror ( "Could not unlink the file." ); }
    else {perror ( "File unlinked." ); }
  }

}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgRmdir()--Remove Directory (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgRmdir(Qlg_Path_Name_T *path,);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgRmdir**() function, like the **rmdir()** function, removes a directory, *path*, provided that the directory is empty; that is, the directory contains no entries other than "dot" (**.**) or "dot-dot" (**..**). The difference is that the **QlgRmdir()** function takes a pointer to a Qlg_Path_Name_T structure, while **rmdir()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of authorities required, return values, usage notes, and related information, see rmdir()--Remove Directory.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the directory to be removed. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- rmdir()--Remove Directory
- QlgMkdir()--Make Directory (using NLS-enabled path name)
- Qp0lUnlink()--Remove Link to File (using NLS-enabled path name)

Example

The following example removes a directory:

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

```
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <Qp0lstdi.h>

main() {

#define mypath_d  "new_dir"
#define mypath_f "new_dir/new_file"

  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path_d;
  struct pnstruct path_f;

  int file_descriptor;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                    */
   /***************************************************************/
  memset((void*)&path_d, 0x00, sizeof(struct pnstruct));
  path_d.qlg_struct.CCSID = 37;
  memcpy(path_d.qlg_struct.Country_ID,US_const,2);
  memcpy(path_d.qlg_struct.Language_ID,Language_const,3);
  path_d.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_d.qlg_struct.Path_Length = sizeof(mypath_d)-1;
  path_d.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_d.pn,mypath_d,sizeof(mypath_d)-1);

  memset((void*)&path_f, 0x00, sizeof(struct pnstruct));
  path_f.qlg_struct.CCSID = 37;
  memcpy(path_f.qlg_struct.Country_ID,US_const,2);
  memcpy(path_f.qlg_struct.Language_ID,Language_const,3);
  path_f.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_f.qlg_struct.Path_Length = sizeof(mypath_f)-1;
  path_d.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_f.pn,mypath_f,sizeof(mypath_f)-1);


  if (QlgMkdir((Qlg_Path_Name_T *)&path_d,S_IRWXU|S_IRGRP|S_IXGRP) !
    perror("QlgMkdir() error");
  else if ((file_descriptor = QlgCreat((Qlg_Path_Name_T *)&path_f,S_IWUSR))
<
    perror("QlgCreat() error");
  else {
    close(file_descriptor);
    QlgUnlink((Qlg_Path_Name_T *)&path_f);
  }

  if (QlgRmdir((Qlg_Path_Name_T *)&path_d) != 0)
    perror("QlgRmdir() error");
```

```
  else
    puts("removed!");
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgSaveStgFree()--Save Storage Free (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>

int QlgSaveStgFree(
  Qlg_Path_Name_T            *Path_Name,
  Qp0l_StgFree_Function_t    *UserFunction_ptr,
  void                       *Function_CtlBlk_ptr);
```

Service Program Name: QP0LLIB3

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

For a description of this function and more information on the parameters, authorities required, return values, error conditions, error messages, usage notes, and related information, see Qp0lSaveStgFree()--Save Storage Free.

API introduced: V5R1

QlgSetAttr()--Set Attributes (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>
int QlgSetAttr
       (Qlg_Path_Name_T      *Path_Name,
        char                 *Buffer_ptr,
        uint                  Buffer_Size,
        uint                  Follow_Symlnk, ...);
```

Service Program Name: QP0LLIB3

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

For a description of this function and information on its parameters, authorities required, return values, error conditions, error messages, usage notes, and related information, see Qp0lSetAttr()--Set Attributes.

API introduced: V5R1

QlgStat()--Get File Information (using NLS-enabled path name)

Syntax

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

int QlgStat(Qlg_Path_Name_T *path,struct stat *buf);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgStat()** function, like the **stat()** function, gets status information about a specified file and places it in the area of memory pointed to by the *buf* argument. The difference is that the **QlgStat()** function takes a pointer to a Qlg_Path_Name_T structure, while **stat()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see stat()--Get File Information.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file from which information is required. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- stat()--Get File Information
- QlgStat64()--Get File Information (large file enabled and using NLS-enabled path name)
- QlgChmod()--Change File Authorizations (using NLS-enabled path name)
- QlgChown()--Change Owner and Group of File (using NLS-enabled path name)
- QlgCreat()--Create or Rewrite File (using NLS-enabled path name)
- QlgLink()--Create Link to File (using NLS-enabled path name)
- QlgLstat()--Get File or Link Information (using NLS-enabled path name)
- QlgMkdir()--Make Directory (using NLS-enabled path name)
- QlgReadlink()--Read Value of Symbolic Link (using NLS-enabled path name)

- QlgSymlink()--Make Symbolic Link (using NLS-enabled path name)
- QlgUtime()--Set File Access and Modification Times (using NLS-enabled path name)
- Qp0lUnlink()--Remove Link to File

Example

The following example gets status information about a file:

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

main() {
  struct stat info;
 #define mypath "/"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                    */
   /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if (QlgStat((Qlg_Path_Name_T *)&path, &info) != 0)
    perror("QlgStat() error");
  else {
    puts("QlgStat() returned the following information about root f/s:")
    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);
  }
}
```

Output: note that the following information will vary from system to system.

```
QlgStat() returned the following information about root f/s:
  inode:   0
 dev id:   1
   mode:   010001ed
  links:   3
    uid:   137
    gid:   500
```

API introduced: V5R1

QlgStat64()--Get File Information (large file enabled and using NLS-enabled path name)

Syntax

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

int QlgStat64(Qlg_Path_Name_T *path, struct stat64 *buf);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The**QlgStat64()** function, like the **stat64()** function, gets status information about a specified file and places it in the area of memory pointed to by the *buf* argument. The difference is that the **QlgStat64()** function takes a pointer to a Qlg_Path_Name_T structure, while **stat64()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see stat64()--Get File Information (Large File Enabled).

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file from which information is required. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- stat()--Get File Information
- stat64()--Get File Information (Large File Enabled)

Example

The following example gets status information about a file:

```
#define _LARGE_FILE_API

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

```
#include <time.h>

main() {
  struct stat64 info;
 #define mypath "/"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                    /* length of the path name or this must */
                    /* be a pointer to the path name.       */
  };
  struct pnstruct path;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath));

  if (QlgStat64((Qlg_Path_Name_T *)&path, &info) != 0)
    perror("QlgStat64() error");
  else {
    puts("QlgStat64() returned the following information about root f/s:");
    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);
  }
}
```

Output: note that the following information will vary from system to system.

```
QlgStat64() returned the following information about root f/s:
  inode:   0
 dev id:   1
   mode:   010001ed
  links:   3
    uid:   137
    gid:   500
```

API introduced: V5R1

QlgStatvfs()--Get File System Information (using NLS-enabled path name)

Syntax

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

int QlgStatvfs(Qlg_Path_Name_T *path, struct statvfs *buf);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgStatvfs()** function, like the **statvfs()** function, gets status information about the file system that contains the file named by the *path* argument. The difference is that the **QlgStatvfs()** function takes a pointer to a Qlg_Path_Name_T structure, while **statvfs()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For moreinformation on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see statvfs()--Get File System Information.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file from which file system information is required. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- statvfs()--Get File System Information
- QlgStatvfs64()--Get File System Information (64-Bit Enabled and using NLS-enabled path name)
- QlgChmod()--Change File Authorizations (using NLS-enabled path name)
- QlgChown()--Change Owner and Group of File (using NLS-enabled path name)
- QlgCreat()--Create or Rewrite File (using NLS-enabled path name)
- QlgLink()--Create Link to File (using NLS-enabled path name)
- QlgUtime()--Set File Access and Modification Times (using NLS-enabled path name)
- Qp0lUnlink()--Remove Link to File

Example

The following example gets status information about a file system:

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

main() {

  struct statvfs info;
 #define mypath "/"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)path name, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if (-1 == QlgStatvfs((Qlg_Path_Name_T *)path name, &info))
    perror("QlgStatvfs() error");
  else {
    puts("QlgStatvfs() 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.

```
QlgStatvfs() 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: V5R1

QlgStatvfs64()--Get File System Information (64-Bit enabled and using NLS-enabled path name)

Syntax

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

int QlgStatvfs64(Qlg_Path_Name_T *path,
                 struct statvfs64 *buf
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgStatvfs64()** function, like the **statvfs64()** function, gets status information about the file system that contains the file named by the *path* argument. The difference is that the **QlgStatvfs64()** function takes a pointer to a Qlg_Path_Name_T structure, while **statvfs64()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see statvfs()--Get File System Information.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file from which file system information is required. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- statvfs()--Get File System Information
- statvfs64()--Get File System Information (64-Bit Enabled)

Example

The following example gets information about a file system.

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

main() {

  struct statvfs info;
 #define mypath "/"

  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100];
       /* This array size must be >= the length */
                        /* of the path name or must be a pointer */
                        /* to the path name.                     */
  };
  struct pnstruct path;

   /*************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                   */
   /*************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-);

  if (-1 == (QlgStatvfs64((Qlg_Path_Name_T *)&path,
                         (struct statvfs64 *)&info)))
  {
    perror("QlgStatvfs64() error");
  }
  else
  {
    puts("QlgStatvfs64() 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);
  }
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgSymlink()--Make Symbolic Link (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgSymlink(
     Qlg_Path_Name_T *pname, Qlg_Path_Name_T *slink);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgSymlink()** function, like the **symlink()** function, creates the symbolic link named by *slink* with the value specified by *pname*. The difference is that the **QlgSymlink()** function takes a pointer to a Qlg_Path_Name_T structure, while **symlink()** takes a pointer to a character string.

Limited information on the **pname* and the **slink* parameter is provided here. For more information on these parameters and for a discussion of authorities required, return values, and related information, see symlink()--Make Symbolic Link.

Parameters

pname

(Input) A pointer to a Qlg_Path_Name_T structure that contains a value or a pointer to a value of the symbolic link. For more information on the Qlg_Path_Name_T structure, see Path name format.

slink

(Input) A pointer to a Qlg_Path_Name_T structure that contains a name or a pointer to a name of the symbolic link to be created. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- symlink()--Make Symbolic Link
- QlgLink()--Create Link to File (using NLS-enabled path name)
- QlgReadlink()--Read Value of Symbolic Link (using NLS-enabled path name)
- Qp0lUnlink()--Remove Link to File

Example

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

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <Qp0lstdi.h>

main() {
  char buf[30];
  int  fd;
 #define mypath_fn "readlink.file"
 #define mypath_sl "readlink.symlink"

  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >=   */
                            /* the length of the path name or  */
                            /* this must be a pointer to the   */
                            /* path name.                      */
  };

  struct pnstruct path_fn;
  struct pnstruct path_sl;
  struct pnstruct path_buf;

   /*****************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                       */
   /*****************************************************************/
  memset((void*)&path_fn, 0x00, sizeof(struct pnstruct));
  path_fn.qlg_struct.CCSID = 37;
  memcpy(path_fn.qlg_struct.Country_ID,US_const,2);
  memcpy(path_fn.qlg_struct.Language_ID,Language_const,3);
  path_fn.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_fn.qlg_struct.Path_Length = sizeof(mypath_fn)-1;
  path_fn.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_fn.pn,mypath_sl,sizeof(mypath_fn)-1);

  memset((void*)&path_sl, 0x00, sizeof(struct pnstruct));
  path_sl.qlg_struct.CCSID = 37;
  memcpy(path_sl.qlg_struct.Country_ID,US_const,2);
  memcpy(path_sl.qlg_struct.Language_ID,Language_const,3);
  path_sl.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_sl.qlg_struct.Path_Length = sizeof(mypath_sl)-1;
  path_sl.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_sl.pn,mypath_sl,sizeof(mypath_sl)-1);

  if ((fd = QlgCreat((Qlg_Path_Name_T *)&path_fn, S_IWUSR))
            < 0)
```

```
    perror("QlgCreat() error");
  else {
    close(fd);
    if (QlgSymlink((Qlg_Path_Name_T *)&path_fn,
                   (Qlg_Path_Name_T *)&path_sl) != 0)
      perror("QlgSymlink() error");

    else {
      if (QlgReadlink((Qlg_Path_Name_T *)&path_sl,
                      (Qlg_Path_Name_T *)&path_buf,
                      sizeof(struct pnstruct))
          < 0)
        perror("QlgReadlink() error");

      else printf("QlgReadlink() returned '%s' for '%s'\n",
                  (Qlg_Path_Name_T *)&path_buf.pn,
                  (Qlg_Path_Name_T *)&path_sl.pn);

      QlgUnlink((Qlg_Path_Name_T *)&path_sl);
    }
    QlgUnlink((Qlg_Path_Name_T *)&path_fn);
  }
}
```

Output:

```
QlgReadlink() returned 'readlink.file' for 'readlink.symlink'
```

API introduced: V5R1

QlgUnlink()--Remove Link to File (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>

int QlgUnlink(Qlg_Path_Name_T *Path_Name);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes for open() API.

The **QlgUnlink()** function, similar to the **unlink()** function, removes a directory entry that refers to a file. **QlgUnlink()**differs from **unlink()** in that the *Path_Name* parameter is a pointer to a Qlg_Path_Name_T structure instead of a pointer to a character string.

For more information on the **Path_Name* parameter and a discussion of the authorities required, return values, and related information, see unlink()--Remove Link to File.

Parameters

Path_Name

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the object to be unlinked. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- unlink()--Remove Link to File
- link()--Create Link to File
- QlgOpen()--Open a File (using NLS-enabled path name)
- QlgRmdir()--Remove Directory (using NLS-enabled path name)

Example

The following example removes a link to a file. This program was stored in a source file with CCSID 37, so the constant string "newfile" will be compiled in CCSID 37. Therefore, the country or region and language specified are United States English, and the CCSID specified is 37.

```
#include <fcntl.h>
#include <stdio.h>
#include <Qp0lstdi.h>

main() {
  const char US_const[3]= "US";
  const char Language_const[4]="ENU";

  struct pnstruct
    {
     Qlg_Path_Name_T  qlg_struct;
     char             pn[7];
    };
  struct pnstruct pns;
  struct pnstruct *pns_ptr = NULL;

  char fn[]="unlink.file";

  memset((void*)&pns, 0x00, sizeof(struct pnstruct));
  pns.qlg_struct.CCSID = 37;
  memcpy(pns.qlg_struct.Country_ID,US_const,2);
  memcpy(pns.qlg_struct.Language_ID,Language_const,3);;
  pns.qlg_struct.Path_Type = 0;
  pns.qlg_struct.Path_Length = sizeof(fn)-1;
  pns.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(pns.pn,fn,sizeof(fn)-1);

  pns_ptr = &pns;

  if (QlgUnlink((Qlg_Path_Name_T *)&pns) != 0)
  {
      perror("QlgUnlink() error");
  }
  else printf("QlgUnlink() successful");
}
```

API introduced: V5R1

QlgUtime()--Set File Access and Modification Times (using NLS-enabled path name)

Syntax

```
#include <utime.h>

int QlgUtime(Qlg_Path_Name_T *path, const struct utimbuf
*times);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgUtime()** function, like the **utime()** function, sets the access and modification times of *path* to the values in the utimbuf structure. The difference is that the **QlgUtime()** function takes a pointer to a Qlg_Path_Name_T structure, while **utime()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see utime()--Set File Access and Modification Times.

Parameters

path

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file for which the times should be changed. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- utime()--Set File Access and Modification Times

Example

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

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

```
#include <fcntl.h>
#include <Qp0lstdi.h>

main() {
  int file_descriptor;
  struct utimbuf ubuf;
  struct stat info;

 #define mypath "utime.file"

  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path;

   /*****************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                       */
   /*****************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if ((file_descriptor =
       QlgCreat((Qlg_Path_Name_T *)&path, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(file_descriptor);
    puts("before QlgUtime()");
    QlgStat((Qlg_Path_Name_T *)&path,&info);
    printf(" utime.file modification time is %ld\n",
           info.st_mtime);
    ubuf.modtime = 0;       /* set modification time to Epoch */
    time(&ubuf.actime);
    if (QlgUtime((Qlg_Path_Name_T *)&path, &ubuf) != 0)
      perror("QlgUtime() error");
    else {
      puts("after QlgUtime()");
      QlgStat((Qlg_Path_Name_T *)&path,&info);
      printf(" utime.file modification time is %ld\n",
             info.st_mtime);
    }
    QlgUnlink((Qlg_Path_Name_T *)&path);
  }
}
```

Output:

```
before QlgUtime()
  utime.file modification time is 749323571
after QlgUtime()
  utime.file modification time is 0
```

API introduced: V5R1

»Perform Miscellaneous File System Functions (QP0FPTOS) API

Required Parameter Group:			
1	Function type	Input	Char(*)
2	Function extension 1	Input	Char(*)
3	Function extension 2	Input	Char(*)

Default Public Authority: *USE

Threadsafe: No

The Perform Miscellaneous File System Function (QP0FPTOS) API is used to perform a variety of file system functions. The first parameter defines the type of function that is requested. Other parameters are optional, depending on the selected function. The output from this API varies, based on the selected function. See the function descriptions for more details.

Authorities and Locks

To call this program you must have *SERVICE special authority, or be authorized to the Service Dump function of Operating System/400 through iSeries Navigator's Application Administration support. The Change Function Usage Information (QSYCHFUI) API, with a function ID of QIBM_SERVICE_DUMP, also can be used to change the list of users allowed to perform dump operations.

Note: Adopted authority is not used.

Required Parameter Group

Required parameters vary according to the selected function. The selected function is identified by the first parameter on the call to the API.

Function Type

INPUT; CHAR(*)

The desired file system function to perform. Valid values follow:

*(1) *DUMP*

Creates a general file system dump in a spooled file with file name "QSYSPRT" and with "QP0FDUMP" in the User Data field. No other parameters are required or supported when *DUMP is specified.

*(2) *DUMPALL*

Creates a variety of file system dumps in a single spooled file with file name "QSYSPRT" and with "QP0FDUMP" in the User Data field. The following table describes the optional

parameter when *DUMPALL is specified.

Function	Function extension 1	Function extension 2	Description
*DUMPALL	Job number (CHAR 6)	(Not supported)	Specifies the job that is dumped. If a job is not specified, the data is dumped for all jobs. If there are multiple jobs with the same number, the first one encountered will be dumped.

*(3) *DUMPLFS*

Creates a dump of logical file system data in a spooled file with file name "QSYSPRT" and with "QP0FDUMP" in the User Data field. The following table describes the optional parameter when *DUMPLFS is specified.

Function	Function extension 1	Function extension 2	Description
*DUMPLFS	Job number (CHAR 6)	(Not supported)	Specifies the job that is dumped. If a job is not specified, the data is dumped for all jobs. If there are multiple jobs with the same number, the first one encountered will be dumped.

*(4) *NFSFORCE*

Sets various values and modes for the network file system. The following table describes the required parameters when *NFSFORCE is specified.

Function	Function extension 1	Function extension 2	Description
*NFSFORCE	V2	ON or OFF	If ON, indicates version 2 mounts only by the client. If QNFSMNTD is started afterwards, then server will permit version 2 mounts only.

*(5) *REBUILDDEVNULL*

Attempts to create the /dev/null and dev/zero character special files. If an existing dev/null or dev/zero object exists that is not a character special file, then the object is renamed to /dev/null.prv or dev/zero.prv. If /dev/null.prv or /dev/zero.prv exists, then it it renamed to /dev/null.prv.001 or /dev/zero.prv.001, /dev/null.prv.002 or /dev/zero.prv.002, and so on, until a name is found for the object. If 999 is exceeded and the rename cannot be done, the object is not renamed and an informational message is issued and the QP0FPTOS program completes successfully. No other parameters are required or supported when *REBUILDDEVNULL is specified.

*(6) *TRACE6ON or *TRACE6OFF*

*TRACE6ON starts the logging of trace messages in the user job log for some network file system functions. *TRACE6OFF stops the logging of these messages.

*(7) *TRACE8ON or *TRACE8OFF*

*TRACE8ON starts the logging of trace messages to the QSYSOPR message queue for some network file system functions. *TRACE8OFF stops the logging of these messages.

*(8) *TRACE9ON or *TRACE9OFF*

*TRACE9ON starts the collection of some network file system statistics and resets the statistics. *TRACE9OFF stops the collection of these statistics.

*(9) *DUMPNFSSTATS*

Creates a file system dump of network file system (NFS) statistics (both client and server) in a spooled file with file name "QSYSPRT" and with "QP0FDUMP" in the User Data field. The information dumped comes from a window of time specified with the *TRACE9ON/OFF function. No other parameters are required or supported when *DUMPNFSSTATS is specified.

Function extension 1

INPUT; CHAR(*)

Function extension 1 is optional or required, based on the first parameter. Whenever it is valid, function extension 1 is described above along with a first parameter description. Function extension 1 is valid when the first parameter is listed below:

*(1) *DUMPALL*

*(2) *DUMPLFS*

*(3) *NFSFORCE*

Function extension 2

INPUT; CHAR(*)

Function extension 2 is optional or required, based on the first parameter. Whenever it is valid, function extension 2 is described above along with a first parameter description. Function extension 2 is valid when the first parameter is listed below:

*(1) *NFSFORCE*

Usage Notes

If this API is called without the first parameter that is required, then message CPFBC53 is issued to the caller. This message specifies a parameter that is not valid. To recover, the caller is pointed to the API documentation.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA0A0 E	Object name already exists.
CPFA0D4 E	File system error occurred. Error number &1.
CPDA0FF E	Program not called. You need *SERVICE authority to call this program.
CPFBC53 E	Invalid parameter.
CPFBC54 E	Not authorized to call program.

Examples

CALL QP0FPTOS *DUMP

CALL QP0FPTOS (*DUMPALL '055229')

CALL QP0FPTOS (*DUMPLFS '055229')

CALL QP0FPTOS (*NFSFORCE V2 ON)

CALL QP0FPTOS *REBUILDDEVNULL

CALL QP0FPTOS *TRACE6ON

CALL QP0FPTOS *TRACE6OFF

CALL QP0FPTOS *TRACE8ON

CALL QP0FPTOS *TRACE8OFF

CALL QP0FPTOS *TRACE9ON

CALL QP0FPTOS *TRACE9OFF

CALL QP0FPTOS *DUMPNFSSTATS

API introduced: V5R2

Qp0lCvtPathToQSYSObjName()-- Resolve Integrated File System Path Name into QSYS Object Name

Syntax

```
#include <qp0lstdi.h>

void Qp0lCvtPathToQSYSObjName(
                              Qlg_Path_Name_T *path_name,
                              void            *qsys_info,
                              char             format_name[8],
                              uint             bytes_provided,
                              uint             desired_CCSID,
                              void            *error_code);
```

Threadsafe: Conditional; see Usage Notes.

The **Qp0lCvtPathToQSYSObjName()** function resolves a given integrated file system path name into the four-part QSYS.LIB or independent ASP QSYS.LIB file system name. The primary three parts of the path name are the following components: library, object, and member. The fourth part of the path name is a character representation of the ASP associated with the object, or the independent ASP name. This depends on whether the path refers to an object in the QSYS.LIB file system or and object in an independent ASP QSYS.LIB file system. If the path contains symbolic links, they will be resolved. If, after symbolic links have been resolved, the path does not refer to an object that could be in either the QSYS.LIB file system or an independent ASP QSYS.LIB file system, the API will return with the error message CPFA0DB indicated in the *error_code* structure. Note that the API does not verify that the object exists.

The API also handles wildcard (*) characters in the path name. If the name or type of a library, object, or member is just an asterisk, *ALL is returned as the name or the type. If an asterisk is part of a library, object, or member name, a name containing an asterisk is returned. For example if the following path name is passed in:

/qsys.lib/test*.file/*.*

the API will return:

- Library name: QSYS
- Library type: *LIB
- Object name: TEST*
- Object type: *FILE
- Member name: *ALL
- Member type: *ALL
- ASP name: *SYSBAS

Note that path name components that follow one containing a wildcard character are ignored.

If less than 8 bytes are supplied for the *error_code* structure, errors will cause an exception to be returned to the caller.

Parameters

path_name

(Input) The path name that refers to the QSYS.LIB »or independent ASP QSYS.LIB file system «object. The path name must refer to an object on the local file system; this API does not recognize file system objects accessed remotely. This path name is in the Qlg_Path_Name_T format. For more information on this structure, see Path name format. If the path_name parameter is NULL or points to invalid storage, a CPFA0CE error message is returned.

qsys_info

(Output) A pointer of type void * that refers to a structure that contains the object name. The format of the data returned is specified by the *format_name* parameter. If the qsys_info parameter is NULL or points to invalid storage, a CPF24B4 error message is returned.

format_name

(Input) An 8-byte character array that indicates how the data will be formatted in the *qsys_info* parameter that is returned. The format is as follows:

QSYS0100

For the format of this structure, see the section Returned Data Format.

If the format_name parameter is NULL or points to invalid storage, a CPF24B4 error message is returned.

bytes_provided

(Input) The number of bytes of data provided in the structure referred to by the *qsys_info* parameter. This value must be at least 8, or a CPF3C24 error message will be returned.

desired_CCSID

(Input) The CCSID the returned object names and types should be converted to. If the value of this parameter is 0, the object names and types will be returned in the job CCSID.

Error code

I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error Code Parameter.

Authorities

Note: Adopted authority is not used.

Authorization Required for the Qp0lCvtPathToQSYSObjName() API

Object Referred to	Authority Required	Message ID
Each directory, preceding the last component, in the path name.	*X	CPFA09C
Object in the QSYS.LIB »or independent ASP QSYS.LIB «file system that the path name refers to.	None	None

Returned Data Format

The following table describes the format of the data returned in the *qsys_info* parameter if the QSYS0100 format is specified. For details on the fields of the structure, see the section Field Descriptions.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Bytes_Returned
4	4	BINARY(4)	Bytes_Available
8	8	BINARY(4)	CCSID_Out
12	C	CHAR(28)	Lib_Name
40	28	CHAR(20)	Lib_Type
60	3C	CHAR(28)	Obj_Name
88	58	CHAR(20)	Obj_Type
108	6C	CHAR(28)	Mbr_Name
136	88	CHAR(20)	Mbr_Type
156	9C	CHAR(28)	Asp_Name

Field Descriptions

ASP Name. The path name component that represents the ASP name, if part of the path, or the ASP that the path is associated with. For paths that refer to objects in independent ASP QSYS.LIB file systems, this will be the name of the ASP device description object. For paths that refer to objects in the QSYS.LIB file system, the value of ASP Name will be *SYSBAS.

Bytes_Available. The total number of bytes required to hold all of the data available in the *qsys_info* parameter.

Bytes_Returned. The number of bytes actually returned in the caller's buffer for the *qsys_info* parameter.

CCSID_Out. The CCSID that the returned text is in. This may be different than the *desired_CCSID* if conversion failed. The text is internally normalized, then converted to the desired CCSID. If this conversion from the normalized form does not succeed, the text will be returned in the CCSID of the normalized form.

Lib_Name. The name of the library that the path name refers to. This field is NULL terminated.

Lib_Type. The type of the object, beginning with an * (asterisk). This field will return either *LIB or *ALL. This field is NULL terminated.

Mbr_Name. The name of the member that the path name refers to. This field is NULL terminated, and could be all NULL (all x'00').

Mbr_Type. The type of the member that the path name refers to. This field is NULL terminated. This field will contain *MBR, *ALL, or all NULL (all x'00').

Obj_Name. The name of the object that the path name refers to. This field is NULL terminated, and could be all NULL (all x'00').

Obj_Type. The type of the object that the path name refers to. This field is NULL terminated. This field could contain an object type (for example *FILE), *ALL, or be NULL (all x'00').

The Lib_Name, Lib_Type, Obj_Name, Obj_Type, Mbr_Name, and Mbr_Type fields of the Qp0l_QSYS_Info_t structure will be filled in as appropriate.

If the object that the path name refers to is a library (*LIB), then the lib_name and lib_type fields will contain that library name and *LIB, respectively, and the Obj_Name and Mbr_Name fields will be NULL (all x'00').

If the object name is not an *FILE object with members, then the Mbr_Name field is NULL (all x'00').

If the object name contains quoted strings, the characters within the strings will not be converted to uppercase.

Error Conditions

None.

Error Messages

CPE3101 E	I/O exception non-recoverable error.
CPE3101 E	I/O exception non-recoverable error.
CPE3418 E	Possible APAR condition or hardware failure.
CPE3474 E	Unknown system state.
CPF24B4 E	Severe error while addressing parameter list.
CPF3BF6 E	Path type value not valid.
CPF3C24 E	Length of the receiver variable is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF9872 E	Program &1 in library &2 ended. Reason code is &3.
CPFA092 E	Path name not converted.
CPFA09C E	Not authorized to object. Object is &1.
CPFA09E E	Object in use. Object is &1.
CPFA09F E	Object damaged. Object is &1.
CPFA0A1 E	An input or output error occurred.
CPFA0A2 E	Information passed to this operation was not valid.
CPFA0A3 E	Path name resolution causes looping.
CPFA0A7 E	Path name too long.
CPFA0A8 E	Operation not allowed in a job running multiple threads.
CPFA0A9 E	Object not found. Object is &1.
CPFA0AA E	Error occurred while attempting to obtain space.
CPFA0AD E	Function not supported by file system.
CPFA0B1 E	Requested operation not allowed. Access problem.
CPFA0C0 E	Buffer overflow occurred.
CPFA0C1 E	CCSID &1 not valid.
CPFA0CE E	Error occurred with path name parameter specified.
CPFA0D4 E	File system error occurred. Error number &1.
CPFA0D9 E	Character string not converted.
CPFA0DB E	Object not a QSYS.LIB object. Object is &1.
CPFA0DD E	Function was interrupted.

CPFA0E0 E	File ID conversion of a directory failed.
CPFA0E1 E	The file ID table is damaged.
CPFA0E2 E	System unable to establish a communications connection to a file server.
CPFA0E4 E	The communications connection with the file server was abnormally ended.
CPFA0E5 E	The communications connection with the file server was abnormally ended.
CPFA0E6 E	Object handle rejected by file server.
CPFA0E7 E	System cannot establish a communications connection with a file server.
CPFA1C5 E	Object is a read only object. Object is &1.

Usage Notes

1. This API will fail and return the error message CPFA0A8 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 file system
     - QSYS.LIB
     - Independent ASP QSYS.LIB
2. This API ignores trailing blank spaces at the end of a path name.

   For example, if the path name is

   ```
   "/qsys.lib/fred.lib/foo.file/abc.mbr      "
   ```

   the trailing blank spaces will be ignored. Thus, the above path name is equivalent to

   ```
   "/qsys.lib/fred.lib/foo.file/abc.mbr"
   ```

Related Information

- The <**qp0lstdi.h**> file (see Header Files for UNIX-Type Functions)
- QlgQp0lCvtPathToQSYSObjName()-- Resolve Integrated File System Path Name into QSYS Object Name

Example

The following example program gets the three-part QSYS name from an integrated file system path name passed to it.

```
#include <qp0lstdi.h>        /* For Qp0lCvtPathToQSYSObjName       */
                             /*     type Qp0l_QSYS_Info_t           */
                             /*     type Qlg_Path_Name_T            */
#include <qusec.h>           /* For type Qus_EC_T                   */
#include <stdlib.h>
#include <stdio.h>

int main ()
{
    /*****************************************************************/
    /* Declaration of path_name parameter                            */
    /*****************************************************************/
    char              path_info_array[500];
    Qlg_Path_Name_T *path_name;
    const char        fname[] =
            "/qsys.lib/jerold.lib/qcsrc.file/testconv.mbr";
    const char        US_const[] = "US";
    const char        Language_const[] = "ENU";
    const char        Path_Name_Del_const[] = "/";

    /*****************************************************************/
    /* Declaration of qsys_info parameter                            */
    /*****************************************************************/
    Qp0l_QSYS_Info_t qsys_info;

    /*****************************************************************/
    /* Declaration of format_name parameter                          */
    /*****************************************************************/
    char format_name[8] = "QSYS0100";

    /*****************************************************************/
    /* Declaration of bytes_provided parameter                       */
    /*****************************************************************/
    uint bytes_provided;

    /*****************************************************************/
    /* Declaration of desired_CCSID parameter.                       */
    /*****************************************************************/
    uint desired_CCSID;

    /*****************************************************************/
    /* Declarations for error_code parameter                         */
    /*****************************************************************/
    Qus_EC_t error_code;
    char     error_string[8];

    /*****************************************************************/
    /* Initialize path_name parameter                                */
    /*****************************************************************/
    memset(path_info_array, 0, sizeof(path_info_array));
    path_name = (Qlg_Path_Name_T *) path_info_array;
```

```
path_name->CCSID = 37;
memcpy(path_name->Country_ID, US_const, 2);
memcpy(path_name->Language_ID, Language_const, 3);
path_name->Path_Type = 0;
path_name->Path_Length = strlen(fname);
memcpy(path_name->Path_Name_Delimiter, Path_Name_Del_const, 1);
memcpy( &(((char *) path_name)[sizeof(Qlg_Path_Name_T)]),
        fname,
        strlen(fname));



/***************************************************************/
/* Initialize qsys_info parameter                              */
/***************************************************************/

/* No initialization requirements for this parameter. */

/***************************************************************/
/* Initialize format_name parameter                            */
/***************************************************************/

/* No additional initialization required. */

/***************************************************************/
/* Initialize bytes_provided parameter.                        */
/***************************************************************/
bytes_provided = sizeof(Qp0l_QSYS_Info_t);

/***************************************************************/
/* Initialize desired_CCSID parameter.                         */
/***************************************************************/
desired_CCSID = 37;

/***************************************************************/
/* Initialize error_code param                                 */
/***************************************************************/
memset(&error_code, 0, sizeof(error_code));
error_code.Bytes_Provided = sizeof(error_code);

/***************************************************************/
/* Call API                                                    */
/***************************************************************/
Qp0lCvtPathToQSYSObjName(path_name,
                        QSYS.LIB_info,
                         format_name,
                         bytes_provided,
                         desired_CCSID,
                        &error_code);

if (error_code.Bytes_Available > 0)
{
    /***********************************************************/
    /* Error occurred.                                         */
    /***********************************************************/
```

```
        printf ("Error occurred: ");
        memcpy (error_string, error_code.Exception_Id, 7);
        error_string[7] = '\0';
        printf ("%s\n", error_string);
        printf ("Bytes available in error code structure: %d.\n",
        error_code.Bytes_Available);
        exit(1);
    }

    /****************************************************************/
    /* API returned successfully.                                   */
    /****************************************************************/

    printf ("Library name: %s\n", qsys_info.Lib_Name);
    printf ("Library type: %s\n", qsys_info.Lib_Type);
    printf ("Object name:  %s\n", qsys_info.Obj_Name);
    printf ("Object type:  %s\n", qsys_info.Obj_Type);
    printf ("Member name:  %s\n", qsys_info.Mbr_Name);
    printf ("Member type:  %s\n", qsys_info.Mbr_Type);
»    printf ("Asp name:     %s\n", qsys_info.Asp_Name);
«    exit(0);
}
```

Output:

```
Library name: JEROLD
Library type: *LIB
Object name:  QCSRC
Object type:  *FILE
Member name:  TESTCONV
Member type:  *MBR
»Asp name:     *SYSBAS
«
```

API introduced: V4R3

Perform File System Operation (QP0LFLOP) API

Required Parameter Group:

1	File System Operation	Input	Binary(4)
2	Input Buffer	Input	Char(*)
3	Length of input buffer	Input	Binary(4)
4	Output Buffer	Output	Char(*)
5	Length of output buffer	Input	Binary(4)
6	Error code	I/O	Char(*)

Default Public Authority: *USE

Threadsafe: No

The Perform File System Operation (QP0LFLOP) API performs miscellaneous file system operations.

Authorities and Locks

The authorities required vary for each operation:

(1) QP0L_RETRIEVE_NETGROUP_FILE_ENTRIES

- The user must have execute (*X) data authority to the /etc directory (if it exists).
- The user must have read (*R) data authority to the /etc/netgroup file (if it exists).

(2) QP0L_WRITE_NETGROUP_FILE_ENTRIES

- The user must have write and execute (*WX) data authority to the /etc directory (if it exists).
- The user must have read and write (*RW) data authority to the /etc/netgroup file (if it exists).

Note: Adopted authority is not used.

Required Parameter Group

The following parameters are required.

File system operation

INPUT; BINARY(4)

The desired file system operation to perform.

You can specify one of the following operations:

(1) QP0L_RETRIEVE_NETGROUP_FILE_ENTRIES

Returns information about all netgroup definitions currently defined in the /etc/netgroup file.

(2) QP0L_WRITE_NETGROUP_FILE_ENTRIES

Recreates the /etc/netgroup file with only the entries provided.

Input buffer

INPUT; CHAR(*)

Information that is required for a given file system operation. The input buffer parameter should be set as follows:

(1) QP0L_RETRIEVE_NETGROUP_FILE_ENTRIES

NULL (no input buffer is required).

(2) QP0L_WRITE_NETGROUP_FILE_ENTRIES

FLOP0200 structure containing the new netgroup entries. For a detailed description of this structure, see Format of FLOP0200 Structure.

Length of input buffer

INPUT;BINARY(4)

The length of the input buffer provided. The length of the input buffer parameter may be specified up to the size of the input buffer area specified by the user program. The length of the input buffer should be 0 when the input buffer is NULL.

Output buffer

OUTPUT; CHAR(*)

Information that is provided by a given file system operation. The output buffer parameter should be set as follows:

(1) QP0L_RETRIEVE_NETGROUP_FILE_ENTRIES

FLOP0100 structure containing enough space to hold all netgroup entries in the /etc/netgroup file. For a detailed description of this structure, see FLOP0100 Structure Description. No partial entries will be returned. To determine if all of the entries were returned, the following semantics will be used:

- If the /etc/netgroup file has no entries defined, bytes available and bytes returned will both be set to 12.
- If the /etc/netgroup file has at least one entry defined, then the bytes available will be greater than 12.
- If all of the defined entries in the /etc/netgroup file could not be returned, then the bytes available will not have the same value as bytes returned.

For example, if the /etc/netgroup file is empty, then bytes available and bytes returned would both be equal to 12. For a different example, if the /etc/netgroup file is not empty, but the length of the output buffer is less than what is required to hold all entries in the /etc/netgroup file, then bytes available would be greater than 12 and bytes returned would be set to 12.

(2) QP0L_WRITE_NETGROUP_FILE_ENTRIES

NULL (no output buffer is required).

Length of output buffer

INPUT; BINARY(4)

The length of the output buffer provided. The length of the output buffer parameter may be specified up to the size of the output buffer area specified by the user program. The length of the output buffer should be 0 when the output buffer is NULL.

Error code

I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error Code Parameter.

Output Buffer Description

The following table describes the order and format of the data returned in the output buffer. For a detailed description of each field, see Field Descriptions.

FLOP0100 Structure Description

This structure is used to return netgroup definitions taken from the /etc/netgroup file.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Number of netgroup entries
These fields repeat for each netgroup entry.		BINARY(4)	Length of netgroup entry
		BINARY(4)	Length of netgroup name
		BINARY(4)	Displacement to member names
		BINARY(4)	Number of member names
		CHAR(*)	Netgroup name
These fields repeat for each member name in the netgroup entry.		BINARY(4)	Length of member name entry
		BINARY(4)	Member name status
		BINARY(4)	Length of member name
		CHAR(*)	Member name

Input Buffer Description

The following table describes the order and format of the data given in the input buffer parameter. For a detailed description of each field, see Field Descriptions.

Format of FLOP0200 Structure

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Number of netgroup entries
These fields repeat for each netgroup entry.		BINARY(4)	Length of netgroup entry
		BINARY(4)	Length of netgroup name
		BINARY(4)	Displacement to member names
		BINARY(4)	Number of member names
		CHAR(*)	Netgroup name
These fields repeat for each member name in the netgroup entry.		BINARY(4)	Length of member name entry
		BINARY(4)	Member name status
		BINARY(4)	Length of member name
		CHAR(*)	Member name

Field Descriptions

Bytes available. The number of bytes of data available to be returned to the user in the output buffer. If all data is returned, bytes available is the same as the number of bytes returned. If the receiver variable was not large enough to contain all of the data, this value is set based on the total number of entries in the /etc/netgroup file.

Bytes returned. The number of bytes of data returned to the user in the output buffer.

Displacement to member names. The offset (in bytes) from the beginning of the netgroup entry to the member names in the netgroup entry.

Length of entry. The length (in bytes) of the current netgroup entry. The length can be used to access the next entry.

Length of member name. The length (in bytes) of the member name.

Length of member name entry. The length (in bytes) of this member name entry.

Length of netgroup name. The length (in bytes) of the netgroup name.

Member name. The member name. This is assumed to be in the CCSID of the job.

Member name status. Describes the type of member name. Possible values follow:

(1) QP0L_MEMBER_IS_A_HOST_NAME

The member name refers to an individual host name.

(2) QP0L_MEMBER_IS_A_NETGROUP_NAME

The member name refers to a netgroup name.

(3) QP0L_MEMBER_IS_AN_IP_ADDRESS

The member name refers to an IP address in the form xxx.xxx.xxx.xxx (for example 123.4.56.78).

Netgroup name. The netgroup name. This is assumed to be in the CCSID of the job.

Number of member names. The number of member names in the netgroup entry.

Number of netgroup entries. The number of complete entries. A value of zero is used if there are no valid entries for the /etc/netgroup file or if the file does not exist.

Usage Notes

The include file for this API is QP0LFLOP.

If none of the required parameters are passed to this API, then message CPFB41F will be issued to the caller. This message lists all of the file operations currently available to the QP0LFLOP API.

WARNING - When the (2) QP0L_WRITE_NETGROUP_FILE_ENTRIES file system operation is requested, the existing /etc/netgroup file will be completely rewritten resulting in a loss of the previous contents of the file.

A netgroup is a way of defining one name (the netgroup name) to represent many other names. The names contained within a netgroup definition are called 'members' of that netgroup. A netgroup member can be either the name of a host system, the name of another netgroup, or an IP address. Netgroup definitions are stored in the /etc/netgroup file and are commonly used by the Network File System (NFS) support when a large group of host systems require common NFS access semantics.

Error Messages

CPFA0D4 E	File system error occurred.
CPE3418 E	Possible APAR condition or hardware failure.
CPF3C90 E	Literal value cannot be changed.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

API introduced: V4R3

Qp0lGetAttr()--Get Attributes

Syntax

```
#include <Qp0lstdi.h<
int Qp0lGetAttr
  (Qlg_Path_Name_T          *Path_Name,
   Qp0l_AttrTypes_List_t    *Attr_Array_ptr,
   char                     *Buffer_ptr,
   uint                      Buffer_Size_Provided,
   uint                     *Buffer_Size_Needed_ptr,
   uint                     *Num_Bytes_Returned_ptr,
   uint                      Follow_Symlnk, ...);
```

Service Program Name: QP0LLIB2

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **Qp0lGetAttr()** function gets one or more attributes, on a single call, for the object that is referred to by the input *Path_Name*. The object must exist, the user must have authority to it, and the requested attributes must be supported by the specific file system. For each requested attribute that is not supported by the file system, **Qp0lGetAttr()** returns zero in the Size of attribute data field, pointed to by the *Buffer_ptr* parameter, for that attribute.

Qp0lGetAttr() either returns the attributes of the symbolic link, or returns the attributes of the object that the symbolic link names. This depends upon the value of the *Follow_Symlnk* parameter.

Qp0lGetAttr() returns all times in seconds since the Epoch so that they are consistent with UNIX-type APIs. The Epoch is the time 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time. If the OS/400 date is set prior to 1970, all time values are zero.

Parameters

Path_Name

(Input) The path name of the object for which attribute information is returned. This path name is in the Qlg_Path_Name_T format. For more information on this structure, see Path name format.

Attr_Array_ptr

(Input) A pointer to a structure listing the requested attributes returned for the object identified by the *Path_Name* parameter. Each entry in the array identifies an attribute, by a constant value, that **Qp0lGetAttr()** returns. The number of requested attributes field must equal the total number of constants. If the **Attr_Array_ptr** is NULL or if the Number of requested attributes field is zero, **Qp0lGetAttr()** returns all the attributes that the API supports that are available for the object. The format of this parameter follows.

Attribute array pointer		
Offset		

Dec	Hex	Type	Field
0	0	BINARY(4)	Number of requested attributes
4	4	ARRAY(*) of BINARY(4)	Array of attribute constants

Array of attribute constants. A list of predefined constants, each identifying a requested attribute. **Qp0lGetattr()** also returns one of these constants in the Attribute identification field, pointed to by the *Buffer_ptr* parameter. The constant must be used to identify the returned attribute because the attributes are returned in any order. Note that the Size of attribute data field, pointed to by the *Buffer_ptr* parameter, contains the total size of data that **Qp0lGetattr()** returns for the constants in this array. Valid values, and sizes of the returned attributes, follow:

0 QP0L_ATTR_OBJTYPE: (CHAR(10)) The object type. See Control Language (CL) information in the iSeries Information center for descriptions of all iSeries object types.

1 QP0L_ATTR_DATA_SIZE: (UNSIGNED BINARY(4)) The size in bytes of the data in this object. This size does not include object headers or the size of extended attributes associated with the object. If this attribute is requested and the size cannot be represented in a BINARY(4) data type, **Qp0lGetAttr()** fails with *errno* [EOVERFLOW]. Refer to QP0L_ATTR_DATA_SIZE_64 for objects whose data sizes are greater than BINARY(4).

2 QP0L_ATTR_ALLOC_SIZE: (UNSIGNED BINARY(4)) The number of bytes that have been allocated for this object. If this size cannot be represented in a BINARY(4) data type, **Qp0lGetAttr()** fails with *errno* [EOVERFLOW]. Refer to QP0L_ATTR_ALLOC_SIZE_64 for objects whose allocated sizes are greater than BINARY(4).

3 QP0L_ATTR_EXTENDED_ATTR_SIZE: (UNSIGNED BINARY(4)) The total number of extended attribute bytes.

4 QP0L_ATTR_CREATE_TIME: (UNSIGNED BINARY(4)) The time the object was created.

5 QP0L_ATTR_ACCESS_TIME: (UNSIGNED BINARY(4)) The time that the object's data was last accessed.

6 QP0L_ATTR_CHANGE_TIME: (UNSIGNED BINARY(4)) The time that the object's data or attributes were last changed.

7 QP0L_ATTR_MODIFY_TIME: (UNSIGNED BINARY(4)) The time that the object's data was last changed.

8 QP0L_ATTR_STG_FREE: (CHAR(1)) Whether the object's data has been moved offline, freeing its online storage. Valid values are:

x'00' QP0L_SYS_NOT_STG_FREE: The object's data is not offline.

x'01' QP0L_SYS_STG_FREE: The object's data is offline.

9 QP0L_ATTR_CHECKED_OUT: Whether an object is checked out or not. When an object is checked out, other users can read and copy the object. Only the user who has the object checked out can change the object. The checkout format is defined in the Qp0lstdi.h header file as data type Qp0l_Checkout_t, and is described in the following table.

Checkout Format			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(1)	Flag indicating whether an object is checked out
1	1	CHAR(10)	User to whom checked out
11	B	CHAR(1)	Reserved
12	C	BINARY(4)	Time checked out

Flag. An indicator as to whether an object is checked out. Valid values are:

x'00' QP0L_NOT_CHECKED_OUT: The object is not checked out.

x'01' QP0L_CHECKED_OUT: The object is checked out.

Reserved. A reserved field. This field must be set to binary zero.

Time checked out. The time the object was checked out. This field represents the number of seconds since the Epoch.

User to whom checked out. The user who has the object checked out. This field is blank if it is not checked out.

10 QP0L_ATTR_LOCAL_REMOTE: (CHAR(1)) Whether an object is stored locally or stored on a remote system. The decision of whether a file is local or remote varies according to the respective file system rules. Objects in file systems that do not carry either a local or remote indicator are treated as remote. Valid values are:

x'01' QP0L_LOCAL_OBJ: The object's data is stored locally.

x'02' QP0L_REMOTE_OBJ: The object's data is on a remote system.

11 QP0L_ATTR_AUTH: The public and private authorities associated with the object.

When the QP0L_ATTR_AUTH attribute is requested, the attribute data is returned in the buffer in the following format. This format is defined in header file Qp0lstdi.h as data type Qp0l_Authority_General_t.

General Authority Format			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(10)	Object owner
10	0A	CHAR(10)	Primary group
20	14	CHAR(10)	Authorization list name
30	1E	CHAR(10)	Reserved
40	28	BINARY(4)	Offset to array of users
44	2C	BINARY(4)	Number of users

48	30	BINARY(4)	Size of user entry field entry
52	34	CHAR(12)	Reserved
		ARRAY(*)	Array of users

Array of users. The names and authorities of the users who are authorized to use the object.

Authorization list name. The name of the authorization list that is used to secure the named object. The value *NONE indicates that no authorization list is used in determining authority to the object.

Number of users. The number of users that are authorized to the object. This is the number of users returned in the array of users.

The QFileSvr.400 file system returns zero for the Number of users and zero for the Offset to array of users. If a primary group is specified, the Network File System (NFS) returns one for the Number of users.

Object owner. The name of the user profile that is the owner of the object or the following special value:

**NOUSRPRF* This special value is used by the Network File System to indicate that there is no user profile on the local iSeries server with a user ID (UID) matching the UID of the remote object.

Offset to array of users. The offset to the names and authorities of the users who are authorized to use the object. This offset is relative to the offset of the QP0L_ATTR_AUTH attribute within the buffer pointed to by the *Buffer_ptr* parameter.

Primary group. The name of the user profile that is the primary group of the object or the following special values:

**NONE* The object does not have a primary group.

**NOUSRPRF* This special value is used by the Network File System to indicate that there is no user profile on the local server with a group ID (GID) matching the GID of the remote object.

Reserved. A reserved field. This field must be set to binary zero.

Size of user entry field entry. The number of bytes returned for each user.

When the QP0L_ATTR_AUTH attribute is requested, the array of users is returned in the buffer in the following format. This format is defined in header file Qp0lstdi.h as data type Qp0l_Authority_Users_t.

Data and Object Authority Format			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(10)	User name
10	0A	CHAR(10)	User data authority
Object rights			
20	14	CHAR(1)	Object management
21	15	CHAR(1)	Object existence
22	16	CHAR(1)	Object alter

23	17	CHAR(1)	Object reference
24	18	CHAR(10)	Reserved
Data rights			
34	22	CHAR(1)	Object operational
35	23	CHAR(1)	Read
36	24	CHAR(1)	Add
37	25	CHAR(1)	Update
38	26	CHAR(1)	Delete
39	27	CHAR(1)	Execute
40	28	CHAR(1)	Exclude
41	29	CHAR(7)	Reserved

Add (*ADD). Authority to add entries to the object. Valid values are:

0 The user does not have add data rights.

1 The user does have add data rights.

Delete (*DELETE). Authority to remove entries from the object. Valid values are:

0 The user does not have delete data rights.

1 The user does have delete data rights.

Execute (*EXECUTE). Authority to run a program or search a library or directory. Valid values are:

0 The user does not have execute data rights.

1 The user does have execute data rights.

Exclude (*EXCLUDE). The user is prevented from accessing the object. Valid values are:

0 The user does not have exclude data rights.

1 The user does have exclude data rights.

Object alter (*OBJALTER). Authority to change the attributes of an object, such as adding or removing triggers for a database file. Valid values are:

0 The user does not have alter object rights.

1 The user does have alter object rights.

Object existence (*OBJEXIST). Authority to control the object's existence and ownership. Valid values are:

0 The user does not have object existence rights.

1 The user does have object existence rights.

Object management (*OBJMGT). Authority to specify security, to move or rename the object, and to add members if the object is a database file. Valid values are:

0 The user does not have object management rights.

1 The user does have object management rights.

Object operational (*OBJOPR). Authority to look at the object's attributes and to use the object as specified by the data authorities that the user has to the object. Valid values are:

0 The user does not have object operational rights.

1 The user does have object operational rights.

Object reference (*OBJREF). Authority to specify the object as the first level in a referential constraint. Valid values are:

0 The user does not have object reference rights.

1 The user does have object reference rights.

Read (*READ). Authority to access the contents of the object. Valid values are:

0 The user does not have read data rights.

1 The user does have read data rights.

Reserved. A reserved field. This field must be set to binary zero.

Update (*UPDATE). Authority to change the content of existing entries in the object. Valid values are:

0 The user does not have update data rights.

1 The user does have update data rights.

User data authority. The operation, use, or access that the user has to an object. Valid values follow:

**RWX* Allows all operations on the object except those that are limited to the owner or controlled by the object rights.

**RW* Allows access to the object attributes and allows the object to be changed. The user cannot use the object.

**WX* Allows use of the object and allows the object to be changed. The user cannot access the object attributes.

**R* Allows access to the object attributes.

**W* Allows the object to be changed.

**X* Allows the use of the object.

**EXCLUDE* All operations on the object are prohibited.

**NONE* Displayed by the system when the user does not have any data authorities.

USER DEF Displayed by the system when the specific data authorities do not match any of the predefined data authority levels above.

User name. The name of a user authorized to use the object. This may be the name of the user profile or one of the following special values:

**NOUSRPRF*	The authorities of either the owner or the primary group of the object for which the profile name could not be determined. This value is used by the Network File System only. It indicates that the user ID (UID) or the group ID (GID) for the remote object does not match any profile on the local iSeries server with that UID or GID.
**NTWIRF*	The authorities of the NetWare Inherited Rights Filter for the object. This value is only used by the QNetWare file system.
**NTWEFF*	The NetWare effective rights to the object. This value is only used by the QNetWare file system.
**PUBLIC*	The authorities of users who are not specifically named and who are not in the object's authorization list.

12 QP0L_ATTR_FILE_ID: (CHAR(16)) An identifier associated with the referred to object. A file ID can be used with Qp0lGetPathFromFileID() to retrieve an object's path name. The file ID is defined in header file Qp0lstdi.h as data type Qp0lFID_t.

13 QP0L_ATTR_ASP: (BINARY(2)) The auxiliary storage pool in which the object is stored.

14 QP0L_ATTR_DATA_SIZE_64: (UNSIGNED BINARY(8)) The size in bytes of the data in this object. This size does not include object headers or the size of extended attributes associated with the object. QP0L_ATTR_DATA_SIZE may be used for objects whose data size can be represented in a BINARY(4) data type.

15 QP0L_ATTR_ALLOC_SIZE_64: (UNSIGNED BINARY(8)) The number of bytes that have been allocated for this object. QP0L_ATTR_ALLOC_SIZE may be used for objects whose allocated size can be represented in a BINARY(4) data type.

16 QP0L_ATTR_USAGE_INFORMATION: Fields indicating how often an object is used. Usage has different meanings according to the specific file system and according to the individual object types supported within a file system. Usage can indicate the opening or closing of a file or can refer to adding links, renaming, restoring, or checking out an object. The usage information format is defined in the Qp0lstdi.h header file as data type Qp0l_Usage_t and is shown in the following table.

Qp0l_Usage_t			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Reset date
4	4	BINARY(4)	Last used date
8	8	BINARY(4)	Days used count
12	C	CHAR(4)	Reserved

Days used count. The number of days an object has been used. Usage has different meanings according to the specific file system and according to the individual object types supported within a file system. Usage can indicate the opening or closing of a file or can refer to adding links, renaming, restoring, or checking out an object. This count is incremented once each day that an object is used and is reset to zero by calling the **Qp0lSetAttr()** API.

Last used date. The number of seconds since the Epoch that corresponds to the date the

object was last used. This field is zero when the object is created. If usage data is not maintained for the OS/400 type or the file system to which an object belongs, this field is zero.

Reserved. A reserved field set to binary zeros.

Reset date. The number of seconds since the Epoch that corresponds to the date the days used count was last reset to zero (0). This date is set to the current date when the **Qp0lSetAttr()** API is called to reset the Days used count to zero.

17 QP0L_ATTR_PC_READ_ONLY: (CHAR(1)) Whether the object can be written to or deleted, have its extended attributes changed or deleted, or have its size changed. Valid values are:

x'00' QP0L_PC_NOT_READONLY: The object can be changed.

x'01' QP0L_PC_READONLY: The object cannot be changed.

18 QP0L_ATTR_PC_HIDDEN: (CHAR(1)) Whether the object can be displayed using an ordinary directory listing.

x'00' QP0L_PC_NOT_HIDDEN: The object is not hidden.

x'01' QP0L_PC_HIDDEN: The object is hidden.

19 QP0L_ATTR_PC_SYSTEM: (CHAR(1)) Whether the object is a system file and is excluded from normal directory searches.

x'00' QP0L_PC_NOT_SYSTEM: The object is not a system file.

x'01' QP0L_PC_SYSTEM: The object is a system file.

20 QP0L_ATTR_PC_ARCHIVE: (CHAR(1)) Whether the object has changed since the last time the file was examined.

x'00' QP0L_PC_NOT_CHANGED: The object has not changed.

x'01' QP0L_PC_CHANGED: The object has changed.

21 QP0L_ATTR_SYSTEM_ARCHIVE: (CHAR(1)) Whether the object has changed and needs to be saved. It is set on when an object's change time is updated, and set off when the object has been saved.

x'00' QP0L_SYSTEM_NOT_CHANGED: The object has not changed and does not need to be saved.

x'01' QP0L_SYSTEM_CHANGED: The object has changed and does need to be saved.

22 QP0L_ATTR_CODEPAGE: (BINARY(4)) The code page derived from the coded character set identifier (CCSID) used for the data in the file or the extended attributes of the directory. If the returned value of this field is zero (0), there is more than one code page associated with the st_ccsid. If the st_ccsid is not a supported system CCSID, the st_codepage is set equal to the st_ccsid.

23 QP0L_ATTR_FILE_FORMAT: (CHAR(1)) The format of the stream file (*STMF). Valid values are:

x'00' QP0L_FILE_FORMAT_TYPE1: The object has the same format as *STMF objects created on releases prior to Version 4 Release 4. It will be saved faster than a *TYPE2 *STMF to releases prior to Version 4 Release 4 of OS/400. It has a mimimum object size of 4096 bytes.

x'01' QP0L_FILE_FORMAT_TYPE2: The object has high performance file access and is a new *STMF object format in Version 4 Release 4 of OS/400. It will be saved slower than a *TYPE1 *STMF to releases prior to Version 4 Release 4 of OS/400. It has a minimum object size of 8192 bytes.

24 QP0L_ATTR_UDFS_DEFAULT_FORMAT: (CHAR(1)) The default file format of stream files (*STMF) created in the user-defined file system. Valid values are:

x'00' QP0L_UDFS_DEFAULT_TYPE1: The stream file (*STMF) has the same format as *STMFs created on releases prior to Version 4 Release 4 of OS/400. It will be saved faster than a *TYPE2 *STMF to releases prior to Version 4 Release 4 of OS/400. It has a mimimum object size of 4096 bytes.

x'01' QP0L_UDFS_DEFAULT_TYPE2: The object has high performance file access and is a new *STMF object format in Version 4 Release 4 of OS/400. It will be saved slower than a *TYPE1 *STMF to releases prior to Version 4 Release 4 of OS/400. It has a minimum object size of 8192 bytes.

25 QP0L_ATTR_JOURNAL_INFORMATION: Journaling information for this object. The journaling information format is defined in the Qp0lstdi.h header file as data type Qp0l_Journal_Info_t and is shown in the following table:

Qp0l_Journal_Info_t			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(1)	Journaling status
1	1	CHAR(1)	Options
2	2	CHAR(10)	Journal identifier (JID)
12	0B	CHAR(10)	Current or last journal name
22	16	CHAR(10)	Current or last journal library name
32	20	BINARY(4)	Last journaling start time

Current or last journal library name. If the value of the journaling status is QP0L_JOURNALED, then this field contains the name of the library containing the currently used journal. If the value of the journaling status is QP0L_NOT_JOURNALED, then this field contains the name of the library containing the last used journal. All bytes in this field will be set to binary zero if this object has never been journaled.

Current or last journal name. If the value of the journaling status is QP0L_JOURNALED, then this field contains the name of the journal currently being used. If the value of the journaling status is QP0L_NOT_JOURNALED, then this field contains the name of the journal last used for this object. All bytes in this field will be set to binary zero if this object has never been journaled.

Journal identifier (JID). This field associates the object being journaled with an identifier that can be used on various journaling-related commands and APIs. This field will be all

binary zeros for recorded byte-stream files.

Journaling status. Current journaling status of the object. This field will be one of the following values:

x'00' QP0L_NOT_JOURNALED: The object is currently not being journaled.

x'01' QP0L_JOURNALED: The object is currently being journaled.

Last journaling start time. The number of seconds since the Epoch that corresponds to the last date and time for which the object had journaling started for it. This field will be set to binary zero if this object has never been journaled.

Options. This field describes the current journaling options. This field is composed of several bit flags and contains one or more of the following bit values:

x'08' QP0L_JOURNAL_SUBTREE: When this flag is returned, this object is a directory with IFS journaling subtree semantics. New objects created within this directory's subtree will inherit the journaling attributes and options from this directory.

x'08' QP0L_JOURNAL_OPTIONAL_ENTRIES: When journaling is active, entries that are considered optional are journaled. The list of optional journal entries varies for each object type. See the Integrated file system topic for information regarding these optional entries for various objects.

x'20' QP0L_JOURNAL_AFTER_IMAGES: When journaling is active, the image of the object after a change is journaled.

x'40' QP0L_JOURNAL_BEFORE_IMAGES: When journaling is active, the image of the object prior to a change is journaled.

26 QP0L_ATTR_ALWCKPWRT: (CHAR(1)) Whether a stream file (*STMF) can be shared with readers and writers during the save-while-active checkpoint processing. Valid values are:

x'00' QP0L_NOT_ALWCKPWRT: The object can be shared with readers only.

x'01' QP0L_ALWCKPWRT: The object can be shared with readers and writers.

27 QP0L_ATTR_CCSID: (BINARY(4)) The CCSID of the data and extended attributes of the object.

28 QP0L_ATTR_SIGNED: (CHAR(1)) Whether an object has an OS/400 digital signature. This attribute is only returned for *STMF objects. Valid values are:

x'00' QP0L_NOT_SIGNED: The object does not have an OS/400 digital signature.

x'01' QP0L_SIGNED: The object does have an OS/400 digital signature.

29 QP0L_ATTR_SYS_SIGNED: (CHAR(1)) Whether the object was signed by a source that is trusted by the system. This attribute is only returned for *STMF objects. Note: this attribute is not returned if the QP0L_ATTR_SIGNED attribute has the value QP0L_NOT_SIGNED. Valid values are:

x'00' QP0L_SYSTEM_SIGNED_NO: (CHAR(1)) None of the signatures came from a source that is trusted by the system.

x'01' QP0L_SYSTEM_SIGNED_YES: The object was signed by a source that is trusted by the system. If the object has multiple signatures, at least one of the signatures came from a source that is trusted by the system.

30 QP0L_ATTR_MULT_SIGS: (CHAR(1)) Whether an object has more than one OS/400 digital signature. This attribute is only returned for *STMF objects. Note: this attribute is not returned if the QP0L_ATTR_SIGNED attribute has the value QP0L_NOT_SIGNED. Valid values are:

x'00' QP0L_MULT_SIGS_NO: The object has only one digital signature.

x'01' QP0L_MULT_SIGS_YES: The object has more than one digital signature. If the QP0L_ATTR_SYS_SIGNED attribute has the value QP0L_SYS_SIGNED, at least one of the signatures is from a source trusted by the system.

31 QP0L_ATTR_DISK_STG_OPT (CHAR(1)) This option should be used to determine how auxiliary storage is allocated by the system for the specified object. This option can only be specified for stream files in the root (/), QOpenSys and user-defined file systems. This option will be ignored for *TYPE1 byte stream files. Valid values are:

x'00' QP0L_STG_NORMAL: The auxiliary storage will be allocated normally. That is, as additional auxiliary storage is required, it will be allocated in logically sized extents to accomodate the current space requirement, and anticipated future requirements, while minimizing the number of disk I/O operations.

x'01' QP0L_STG_MINIMIZE: The auxiliary storage will be allocated to minimize the space used by the object. That is, as additional auxiliary storage is required, it will be allocated in small sized extents to accomodate the current space requirement. Accessing an object composed of many small extents may increase the number of disk I/O operations for that object.

x'02' QP0L_STG_DYNAMIC: The system will dynamically determine the optimum auxiliary storage allocation for the object, balancing space used versus disk I/O operations. For example, if a file has many small extents, yet is frequently being read and written, then future auxiliary storage allocations will be larger extents to minimize the number of disk I/O operations. Or, if a file is frequently truncated, then future auxiliary storage allocations will be small extents to minimize the space used. Additionally, information will be maintained on the stream file sizes for this system and its activity. This file size information will also be used to help determine the optimum auxiliary storage allocations for this object as it relates to the other objects sizes.

32 QP0L_ATTR_MAIN_STG_OPT: (CHAR(1)) This option should be used to determine how main storage is allocated and used by the system for the specified object. This option can only be specified for stream files in the root (/), QOpenSys and user-defined file systems. Valid values are:

x'00' QP0L_STG_NORMAL: The main storage will be allocated normally. That is, as much main storage as possible will be allocated and used. This minimizes the number of disk I/O operations since the information is cached in main storage.

x'01' QP0L_STG_MINIMIZE: The main storage will be allocated to minimize the space used by the object. That is, as little main storage as possible will be allocated and used. This minimizes main storage usage while increasing the number of disk I/O operations since less information is cached in main storage.

x'02' QP0L_STG_DYNAMIC: The system will dynamically determine the optimum main storage allocation for the object depending on other system activity and main storage contention. That is, when there is little main storage contention, as much storage as possible will be allocated and used to minimize the number of disk I/O operations. And when there is significant main storage contention, less main storage will be allocated and used to minimize the main storage contention. This option only has an effect when the storage pool's paging option is *CALC. When the storage pool's paging option is *FIXED, the behavior is the same as QP0L_STG_NORMAL. When the object is accessed thru a file server, this option has no effect. Instead, its behavior is the same as QP0L_STG_NORMAL.

33 QP0L_ATTR_DIR_FORMAT: (CHAR(1)) The format of the specified directory object. Valid values are:

x'00' QP0L_DIR_FORMAT_TYPE1: The directory of type *DIR has the original directory format. The Convert Directory (CVTDIR) command may be used to convert from the *TYPE1 format to the *TYPE2 format.

x'01' QP0L_DIR_FORMAT_TYPE2: The directory of type *DIR is optimized for performance, size, and reliability compared to directories having the *TYPE1 format.

34 QP0L_ATTR_AUDIT: (CHAR(10)) The auditing value associated with the object. Valid values are:

**NONE* No auditing occurs for this object when it is read or changed regardless of the user who is accessing the object.

**USRPRF* Audit this object only if the current user is being audited. The current user is tested to determine if auditing should be done for this object. The user profile can specify if only change access is audited or if both read and change accesses are audited for this object.

**CHANGE* Audit all change access to this object by all users on the system.

**ALL* Audit all access to this object by all users on the system. All access is defined as a read or change operation.

300 QP0L_ATTR_SUID: (CHAR(1)) Set effective user ID (UID) at execution time. This value is ignored if the specified object is a directory. Valid values are:

x'00' QP0L_SUID_OFF: The user ID (UID) is not set at execution time.

x'01' QP0L_SUID_ON: The object owner is the effective user ID (UID) at execution time.

301 QP0L_ATTR_SGID: (CHAR(1)) Set effective group ID (GID) at execution time. Valid values are:

x'00' QP0L_SGID_OFF: If the object is a file, the group ID (GID) is not set at execution time. If the object is a directory in the root ('/'), QOpenSys, and user-defined file systems, the group ID (GID) of objects created in the directory is set to the effective GID of the thread creating the object. This value cannot be set for other file systems.

x'01' QP0L_SGID_ON: If the object is a file, the group ID (GID) is set at execution time. If the object is a directory, the group ID (GID) of objects created in the directory is set to the GID of the parent directory.

Number of requested attributes. The total number of requested attributes that **Qp0lGetAttr()** returns. This field is required when the *Attr_Array_ptr* parameter is not NULL and must equal the number of constants in the array to which it points. When this field is zero, **Qp0lGetAttr()** returns all the attributes that are supported by the API and that are available for the object.

Buffer_ptr

(Input) A pointer to a buffer that the caller allocates for **Qp0lGetAttr()** to return the requested data. The caller also sets the *Buffer_Size_Provided* parameter to the number of bytes that are allocated for this buffer.

If the buffer provided is not large enough to hold all of the requested data, **Qp0lGetAttr()** fills the buffer with as much data as possible and sets the value pointed to by the *Buffer_Size_Needed_ptr* parameter equal to the number of bytes required for all of the requested data to be returned.

When the *Buffer_ptr* is NULL, **Qp0lGetAttr()** returns the total number of bytes needed to hold all of the requested attributes and sets the *Buffer_Size_Needed_ptr* parameter to point to this value.

Qp0lGetAttr() identifies each entry that it returns in the buffer with the constant that the user supplied in the input structure pointed to by the *Attr_Array_ptr* parameter. **Qp0lGetAttr()** returns this constant in the Attribute identification field. The constant must be used to identify the returned attribute because the attributes are returned in any order.

Qp0lGetAttr() fills the buffer with an entry for each requested attribute in the following format:

Buffer Pointer			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Offset to next attribute entry
4	4	BINARY(4)	Attribute identification
8	8	BINARY(4)	Size of attribute data
12	C	CHAR(4)	Reserved
16	10	CHAR(*)	Attribute data

Attribute data. The attribute data that was requested.

Attribute identification. The constant that identifies the returned attribute. Valid values follow and are the same constants as provided by the caller of **Qp0lGetAttr()**, pointed to by the *Attr_Array_ptr* parameter.

See the *Attr_Array_ptr* parameter for descriptions of each of these attribute values.

0	QP0L_ATTR_OBJTYPE
1	QP0L_ATTR_DATA_SIZE
2	QP0L_ATTR_ALLOC_SIZE
3	QP0L_ATTR_EXTENDED_ATTR_SIZE
4	QP0L_ATTR_CREATE_TIME
5	QP0L_ATTR_ACCESS_TIME
6	QP0L_ATTR_CHANGE_TIME
7	QP0L_ATTR_MODIFY_TIME
8	QP0L_ATTR_STG_FREE
9	QP0L_ATTR_CHECKED_OUT
10	QP0L_ATTR_LOCAL_REMOTE
11	QP0L_ATTR_AUTH
12	QP0L_ATTR_FILE_ID
13	QP0L_ATTR_ASP
14	QP0L_ATTR_DATA_SIZE_64
15	QP0L_ATTR_ALLOC_SIZE_64
16	QP0L_ATTR_USAGE_INFORMATION
17	QP0L_ATTR_PC_READ_ONLY
18	QP0L_ATTR_PC_HIDDEN
19	QP0L_ATTR_PC_SYSTEM
20	QP0L_ATTR_PC_ARCHIVE
21	QP0L_ATTR_SYSTEM_ARCHIVE
22	QP0L_ATTR_CODEPAGE
23	QP0L_ATTR_FILE_FORMAT
24	QP0L_ATTR_UDFS_DEFAULT_FORMAT
25	QP0L_ATTR_JOURNAL_INFORMATION
26	QP0L_ATTR_ALWCKPWRT
27	QP0L_ATTR_CCSID

28 QP0L_ATTR_SIGNED

29 QP0L_ATTR_SYS_SIGNED

30 QP0L_ATTR_MULT_SIGS

31 QP0L_ATTR_DISK_STG_OPT

32 QP0L_ATTR_MAIN_STG_OPT

33 QP0L_ATTR_DIR_FORMAT

34 QP0L_ATTR_AUDIT

300 QP0L_ATTR_SUID

301 QP0L_ATTR_SGID

Offset to next attribute entry. The offset to the next attribute entry in the buffer. This offset is relative to the start of the buffer. An offset of zero means that no more attribute entries follow.

Reserved. A reserved field set to binary zero.

Size of attribute data. The total size of all the data for this attribute. The special value of 0 in this field indicates that the attribute is not supported by the file system in which the object is stored. The attribute data is padded with hexadecimal zeros. The size indicated in this field does not include the padding bytes.

Buffer_Size_Provided

(Input) The number of bytes the caller allocates in a buffer for the return of requested data. The buffer is pointed to by the *Buffer_ptr* parameter.

If this size is set to zero or is not large enough to hold all of the requested data, **Qp0lGetAttr()** fills the buffer with as much data as possible and sets the value pointed to by the *Buffer_Size_Needed_ptr* parameter equal to the number of bytes required for all of the requested data to be returned.

Buffer_Size_Needed_ptr

(Output) A pointer to the number of bytes that the caller needs to allocate for **Qp0lGetAttr()** to return all of the requested data.

Num_Bytes_Returned_ptr

(Output) A pointer to the actual number of bytes of data returned in the user buffer. This field is zero if the *Buffer_ptr* parameter is NULL.

Follow_Symlnk

(Input) If the last component in the *Path_Name* is a symbolic link, this parameter determines if the symbolic link or the path contained in the symbolic link is acted upon: Valid values are:

0 QP0L_DONOT_FOLLOW_SYMLNK: A symbolic link in the last component is not followed. Attributes of the symbolic link object are returned.

1 QP0L_FOLLOW_SYMLNK: A symbolic link in the last component is followed. The attributes of the object contained in the symbolic link are returned.

Authorities

Note: Adopted authority is not used.

Authorization Required for Qp0lGetAttr()		
Object Referred to	**Authority Required**	***errno***
Each directory, preceding the last component, in the *Path_Name*	*X	EACCES
Object, when retrieving the QP0L_ATTR_AUTH attribute	*OBJMGT	EACCES
Note: If the file system supports *ALLOBJ special authority and if you have *ALLOBJ special authority, you do not need the listed object authority.		

Return Value

0 **Qp0lGetAttr()** was successful.

-1 **Qp0lGetAttr()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **Qp0lGetAttr()** is not successful, *errno* indicates one of the following errors:

[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.

[ECANCEL]

Operation canceled.

[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.

[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.

[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.

[EOFFLINE]

Object 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 successfully restored. 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.

[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.

[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.
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

   Qp0lGetAttr() could return zero for the QP0L_ATTR_ACCESS_TIME value (in the buffer area) under some conditions.

   Refer to the CL Programming book for more information regarding which object types maintain

usage information that is returned for the QP0L_ATTR_USAGE_INFORMATION attribute.

When **Qp0lGetAttr()** is performed on a physical file member, the QP0L_ATTR_JOURNAL_INFORMATION attribute will contain journaling information applicable to the physical file that contains the member.

Related Information

- The <**Qp0lstdi.h**> file (see Header Files for UNIX-Type Functions)
- The <**qlg.h**> file (see Header Files for UNIX-Type Functions)
- »chmod()--Change File Authorizations «
- fstat()--Get File Information by Descriptor
- lstat()--Get File or Link Information
- QlgGetAttr()--Get Attributes (using NLS-enabled path name)
- QlgStat()--Get File Information (using NLS-enabled path name)
- QlgLstat()--Get File or Link Information (using NLS-enabled path name)
- »Qp0lSetAttr()--Set Attributes«
- stat()--Get File Information

Example

Following is an example showing a call to **Qp0lGetAttr()**. The example also shows a call to **Qp0lSaveStgFree()**.

See Code disclaimer information for information pertaining to code examples.

```
/*****************************************************************/
#include "Qp0lstdi.h"
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/types.h>
#include <qusec.h>
#include <time.h>

int Save(Qp0l_Pathnames_t *Path_name_ptr)
{
  /**************************************************************/
```

```
  /* No function here in the example                                  */
  /********************************************************************/
};

void SaveAnObject(Qp0l_Pathnames_t *Path_name_ptr,
                  int              *Return_code_ptr,
                  int              *Return_value_ptr,
                  void             *Function_CtlBlk_ptr)
{
  /********************************************************************/
  /* This function saves a file and its hard links to tape.           */
  /********************************************************************/
  int rc;

  if ((Path_name_ptr == (Qp0l_Pathnames_t *)NULL) ||
      (Path_name_ptr->Number_Of_Names == 0))
  {
    printf("In User Exit Program with null Path \n");
  }
  else
  {
    /* This example calls a function (Save) that could call the  */
    /* Save Object (QsrSave) API.  The QsrSave API is designed to */
    /* save a copy of one or more objects that can be used in the */
    /* integrated file system.  For details on using QsrSave, see  */
    /* the Backup and Recovery API part.                           */

    rc = (Save(Path_name_ptr));

    *Return_code_ptr = rc;
    *Return_value_ptr = errno;

    if (rc == 0)
    {
      /* Other processing for a successfully saved object. */
    }
    else
    {
      /* Optional processing such as storing information   */
      /* to be returned to the caller in the function      */
      /* control block area, or building a list of the     */
      /* files whose save attempts failed, or other.       */
    }
  }
  return;
}                               /* end SaveAnObject exit program */

int main (int argc, char *argv[])
{
#define MYPN "ADIR/ASTMF"
  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  const char Path_Name_Del_const[2] = "/";

  struct pnstruct
  {
    Qlg_Path_Name_T  qlg_struct;
    char             pn[1];
```

```
};
struct pnstruct pns;
struct pnstruct *pns_ptr = NULL;

struct attrStruct
{
  Qp0l_AttrTypes_List_t  attr_struct;
  uint AttrTypes[10];
};
struct attrStruct Attr_types_ptr;
Qp0l_Attr_Header_t *attrPtr;
char *attrValp;

Qp0l_StgFree_Function_t  User_function;

struct
{
  uint    AnyData_to_the_exitprogram;
  uint    AnyData_not_processed_by_the_API;
} CtlBlkAreaName;

time_t mytime;
char BufferArea[250];
unsigned int buff_size_provided;
unsigned int buff_size_needed = 0;
unsigned int num_bytes_returned = 0;
unsigned int follow_sym;
int done=0;
int rc;
int returned_data_index = 0;

/**************************************************************/
/* Initialize Get Attributes Parameters                       */
/**************************************************************/
memset((void*)&pns, 0x00, sizeof(struct pnstruct));
pns.qlg_struct.CCSID = 37;
memcpy(pns.qlg_struct.Country_ID,US_const,2);
memcpy(pns.qlg_struct.Language_ID,Language_const,3);
pns.qlg_struct.Path_Type = 0;
pns.qlg_struct.Path_Length = sizeof(MYPN)-1;
memcpy(pns.qlg_struct.Path_Name_Delimiter,Path_Name_Del_const,1);
memcpy(pns.pn,MYPN,sizeof(MYPN));
memset((void *)&Attr_types_ptr, 0x00,sizeof(struct attrStruct));
pns_ptr = &pns;

Attr_types_ptr.attr_struct.Number_Of_ReqAttrs = 2;
Attr_types_ptr.AttrTypes[0] = QP0L_ATTR_ACCESS_TIME;
Attr_types_ptr.AttrTypes[1] = QP0L_ATTR_STG_FREE;

buff_size_provided = 250;

follow_sym = QP0L_FOLLOW_SYMLNK;

/**************************************************************/
/* Call the Qp0lGetAttr() API to retrieve attributes to       */
/* determine if selection criteria can be met for calling     */
/* the Qp0lSaveStgFree() API.                                 */
/**************************************************************/
```

```
rc = Qp0lGetAttr((Qlg_Path_Name_T *)&pns,
                 (Qp0l_AttrTypes_List_t *)&Attr_types_ptr,
                 BufferArea,
                 buff_size_provided,
                 &buff_size_needed,
                 &num_bytes_returned,
                 follow_sym);
if (rc == 0)                      /* check API return code */
{
  /* Must first check if any data was returned. */
  if (num_bytes_returned > 0)
  {
    attrPtr = (Qp0l_Attr_Header_t *)BufferArea;
    while(!done)
    {
      attrValp = (char *)attrPtr +
        sizeof(Qp0l_Attr_Header_t);  /* Point to attr value   */
      /*****************************************************/
      /* The following code prints the two attributes that  */
      /* were returned.  Add more code here, for example,   */
      /* to determine if the returned attributes meet       */
      /* the criteria or policies for storage freeing.      */
      /*****************************************************/
      printf ("****************************************\n");
      printf ("Attr ID #%d = %d - ",
              returned_data_index,
              attrPtr->Attr_ID);
      if(attrPtr->Attr_Size > 0)
      {
        switch (attrPtr->Attr_ID)
        {
          case QP0L_ATTR_ACCESS_TIME:
              printf("QP0L_ATTR_ACCESS_TIME\n");
              memcpy((void *)&mytime,
                     (void *)attrValp,
                     attrPtr->Attr_Size);
              printf ("%s", ctime(&mytime));
              break;
          case QP0L_ATTR_STG_FREE:
              printf ("QP0L_ATTR_STG_FREE\n");
              switch (attrValp[0])
              {
                case QP0L_SYS_STG_FREE:
                    printf ("--Is storage freed--\n");
                    break;
                case QP0L_SYS_NOT_STG_FREE:
                    printf ("--Is not storage freed--\n");
                    break;
                default:
                    printf ("Invalid data: %d.\n",
                            attrValp[0]);
                    break;
              }
              break;
          default:
              printf ("Undefined return type (attr id unknown.)\n");
              break;
        }                      /* end switch                    */
```

```
        }
        else
          printf("Attribute has no value\n");
        printf("***Size of this attr's data: %d\n",
               attrPtr->Attr_Size);
        printf("***Offset to next attr: %d\n",
               attrPtr->Next_Attr_Offset);
        ++returned_data_index;
        if(attrPtr->Next_Attr_Offset > 0) /* If more data        */
          attrPtr = (Qp0l_Attr_Header_t *) /* Set attribute      */
            &(BufferArea[attrPtr->Next_Attr_Offset]); /* pointer */
        else                     /* No more data                 */
          done = 1;              /* End the loop                 */
      }

    /***********************************************************/
    /* Initialize Save Storage Free Parameters.  The path      */
    /* name parameter was already initialized as part of the   */
    /* call to Qp0lGetAttr() API and is assumed, in this       */
    /* example, to be the same pathname.  Both APIs require    */
    /* the same path name format.                              */
    /***********************************************************/
      memset((void *)&User_function,0x00,sizeof(Qp0l_StgFree_Function_t));
      User_function.Mltthdacn[0] = QP0L_MLTTHDACN_NOMSG;
      User_function.Function_Type = QP0L_USER_FUNCTION_PTR;
      User_function.Procedure = &SaveAnObject;

      rc = Qp0lSaveStgFree((Qlg_Path_Name_T *)&pns,
                           &User_function,
                           &CtlBlkAreaName);
      if(rc == 0)
        printf("Qp0lSaveStgFree() Successful!");
      else
      {/* Unsuccessful return from Qp0lSaveStgFree() API.  */
      /* The following code prints the errno value message.  */
        rc = errno;
        printf("ERROR on Qp0lSaveStgFree(): error = %d\n", rc);
        perror("Error message");
      }
    }                            /* if (num_bytes_returned > 0)   */
    else
      rc = EUNKNOWN;
  }                  /* end rcGA == 0, Qp0lGetAttr() was successful */
  else
  {
    rc = errno;
    printf("ERROR on Qp0lGetAttr(): error = %d\n", rc);
    perror("Error message");
  }
  return(rc);
} /* end main */
```

API introduced: V4R3

Qp0lGetPathFromFileID()--Get Path Name of Object from Its File ID

Syntax

```
#include <Qp0lstdi.h>

char *Qp0lGetPathFromFileID(char *buf, size_t size,
                            Qp0lFID_t fileid);
```

Threadsafe: Yes

The **Qp0lGetPathFromFileID()** function determines an absolute path name of the file identified by *fileid* and stores it in *buf*. The components of the returned path name are not symbolic links. If the file has more than one path name, only one is returned.

The access time of each directory in the absolute path name of the file (excluding the file itself) is updated.

If *buf* is a NULL pointer, **Qp0lGetPathFromFileID()** returns a NULL pointer and the EINVAL error.

The contents of *buf* after an error are not defined.

Qp0lGetPathFromFileID() is supported in the root (/), QOpenSys, and user-defined file systems.

Parameters

buf

(Output) A pointer to a buffer that will be used to hold an absolute path name of the file identified by *fileid*. The buffer must be large enough to contain the full path name including the terminating NULL character.

The path name 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 QlgGetPathFromFileID()--Get Path Name of Object from Its File ID (using NLS-enabled path name) for a description and an example of supplying the *buf* in any CCSID.

size

(Input) The number of bytes in the buffer buf.

fileid

(Input) The identifier of the file whose path name is to be returned. This identifier is logged in audit journal entries to identify the file being audited. See the Parent File ID and Object File ID fields of the audit journal entries described in the iSeries Security Reference book.

Authorities

Note: Adopted authority is not used.

Authorization required for Qp0lGetPathFromFileID()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the file	*RX	EACCES
The file itself	*R	EACCES

Return Value

value

Qp0lGetPathFromFileID() was successful. The value returned is a pointer to *buf*.

NULL

Qp0lGetPathFromFileID() was not successful. The *errno* global variable is set to indicate the error. After an error, the contents of *buf* are not defined.

Error Conditions

If Qp0lGetPathFromFileID() 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.

[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.

[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.

[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.

No path names were found for this *fileid* or the user is not authorized to any of the paths.

[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.

[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.

[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. File System Differences

   The following file systems do not support **Qp0lGetPathFromFileID()**:

   - Network File System
   - QSYS.LIB
   - Independent ASP QSYS.LIB
   - QDLS
   - QOPT
   - QFileSvr.400
   - QNetWare
   - QNTC

Related Information

- The <**Qp0lstdi.h**> file (see Header Files for UNIX-Type Functions)
- QlgGetPathFromFileID()--Get Path Name of Object from Its File ID (using NLS-enabled path name)

Example

The following example determines the path name of a file, given its file ID. In this example, the fileid is hardcoded. More realistically, the fileid is obtained from the audit journal entry and passed to **Qp0lGetPathFromFileID()**.

```
#include <Qp0lstdi.h>
#include <stdio.h>

main()
{
  char        path[1024];
  Qp0lFID_t   fileid = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x80, 0xFF, 0xCF, 0x00};

  if (Qp0lGetPathFromFileID(path, sizeof(path), fileid) == NULL)
    perror("Qp0lGetPathFromFileID() error");
  else
    printf("The file's path is: %s\n", path);
}
```

Output:

```
The file's path is: /myfile
```

API introduced: V3R1

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

Qp0lOpen()--Open File

Syntax

```
#include <Qp0lstdi.h>

int Qp0lOpen(Qlg_Path_Name_T *Path_Name,
int oflag, . . .);
```

Theadsafe: Conditional; see Usage Notes on open() API.

The **Qp0lOpen()** function, similar to the **open()** function, opens a file and returns a number called a **file descriptor**. **Qp0lOpen()**differs from **open()** in that the *Path_Name* parameter is a pointer to a Qlg_Path_Name_T structure instead of a pointer to a character string.

Only the *Path_Name* parameter is described here. For a discussion of the other parameters, authorities required, return values, and related information, see open()--Open File.

Note: To use this API with large file APIs, you must specify the O_LARGEFILE flag on the *oflag* parameter.

Parameters

Path_Name

(Input) The path name of the file to be opened. This path name is in the Qlg_Path_Name_T format. For more information on this structure, see Path Name Format.

Related Information

- The <**fcntl.h**> file (see Header Files for UNIX-Type Functions)
- open()--Open File
- close()--Close File or Socket Descriptor

Example

The following example creates and opens an output file for exclusive access. This program was stored in a source file with CCSID 37, so the constant string "newfile" will be compiled in coded character set identifier (CCSID) 37. Therefore, the country or region and language specified are United States English, and the CCSID specified is 37.

```
#include <fcntl.h>
#include <stdio.h>
#include <Qp0lstdi.h>
```

```
main()
{
  int fildes;

  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  const char Path_Name_Del_const[2] = "/";

  struct pnstruct
    {
     Qlg_Path_Name_T  qlg_struct;
     char             pn[7];
    };
  struct pnstruct pns;
  struct pnstruct *pns_ptr = NULL;

  char fn[]="newfile";

  memset((void*)&pns, 0x00, sizeof(struct pnstruct));
  pns.qlg_struct.CCSID = 37;
  memcpy(pns.qlg_struct.Country_ID,US_const,2);
  memcpy(pns.qlg_struct.Language_ID,Language_const,3);;
  pns.qlg_struct.Path_Type = 0;
  pns.qlg_struct.Path_Length = sizeof(fn) - 1;
  memcpy(pns.qlg_struct.Path_Name_Delimiter,
         Path_Name_Del_const,1);
  memcpy(pns.pn,fn,sizeof(fn));

  pns_ptr = &pns;
  if(fildes = QpOlOpen((Qlg_Path_Name_T *)pns_ptr,
           O_WRONLY|O_CREAT|O_EXCL, S_IRWXU)) == -1)
  {
        perror("QpOlOpen() error");
  }

}
```

API introduced: V4R4

Qp0lProcessSubtree()--Process a Path Name

Syntax

```
 #include <Qp0lstdi.h>

 int Qp0lProcessSubtree (
  Qlg_Path_Name_T           *Path_Name,
  uint                       Subtree_level,
  Qp0l_Objtypes_List_t      *Objtypes_array_ptr,
  uint                       Local_remote_obj,
  Qp0l_IN_EXclusion_List_t *IN_EXclusion_ptr,
  uint                       Err_recovery_action,
  Qp0l_User_Function_t      *UserFunction_ptr,
  void                      *Function_CtlBlk_ptr, ...);
```

Service Program Name: QP0LLIB2

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **Qp0lProcessSubtree()** function searches the directory tree under a specific path name. It selects and passes objects, one at a time, to an exit program that is identified on its call. The exit program can be either a procedure or a program.

Qp0lProcessSubtree() performs recursive read operations to access any object in any file system. The order in which objects are selected and passed to the exit program can vary within a given file system and within a given directory, dependent on file system rules. The only guaranteed ordering is that all selected objects within a given directory are passed to the exit program before the parent directory is passed to the exit program.

Parameters

Path_Name

(Input) The path name where **Qp0lProcessSubtree()** starts its search. All relative path names are relative to the current directory at the time of the call to **Qp0lProcessSubtree()**. This path name is in the Qlg_Path_Name_T format. For more information on this structure, see Path Name Format.

The *Path_Name* parameter must be NULL to use the *IN_EXclusion_ptr* parameter to enter multiple path names for inclusion on a single call to **Qp0lProcessSubtree**().

Subtree_level

(Input) An unsigned integer that tells **Qp0lProcessSubtree()** whether or not to open subdirectories in the path being processed. Valid values follow:

0

QP0L_SUBTREE_YES: All subdirectories are opened by **Qp0lProcessSubtree()** so that the objects they contain are sent to the exit program if they meet the caller's selection

criteria.

1

QP0L_SUBTREE_NO: Only first-level objects are processed. The names of subdirectories, which meet the selection criteria, are passed to the exit program, but they are not opened by **Qp0lProcessSubtree()**. Thus, the objects the subdirectories contain are not matched against selection criteria and therefore are not sent to the exit program.

Objtypes_array_ptr

(Input) A pointer to an array of object types. Each entry in the array identifies an object type that **Qp0lProcessSubtree()** » uses to determine what will be passed to the exit program. The « Number of object types field contains the total number of object types in the array. A NULL pointer means that there is no filtering according to object type and that all object types that meet other selection criteria are passed to the exit program.

The structure for this parameter follows.

Object Types ArrayPointer

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Number of object types
4	4	ARRAY(*) of CHAR(11)	Array of object types structure

Array of object types structure

An array identifying each object type » used to determine what will be passed to the exit program « when processing a path. Each entry is limited to 11 characters, including a NULL terminator, and is padded with blanks. Object types must be entered in standard OS/400 object type format which is all capital letters, preceded by an asterisk (*). For a complete list of the available object types, see Object Types in the CL topic.

Qp0lProcessSubtree() verifies that valid OS/400 object types are entered and returns the *errno* EINVAL when an object type that is not valid is entered. Although some object types are scoped to a specific file system, **Qp0lProcessSubtree()** does not validate object types according to file systems.

Valid special values for this parameter follow:

ALLDIR:

Select all directory object types. This includes *LIB, *DIR, *FLR, *FILE, and *DDIR object types.

»***ALLQSYS:***

Select all QSYS.LIB object types. This includes all objects in the QSYS.LIB file system and all independent ASP QSYS.LIB file systems which are available when the API is first called.

Note: *IN_EXclusion_ptr* must also be specified as an inclusion array. If *NOQSYS is specified, *ALLQSYS cannot also be specified.«

ALLSTMF:

Select all OS/400 stream file object types. This includes *MBR, *DOC, *STMF, *DSTMF, and *USRSPC object types.

MBR:

Select all OS/400 database file member types.

»***NOQSYS:***

Exclude all QSYS.LIB object types. This includes all objects in the QSYS.LIB file system and all independent ASP QSYS.LIB file systems which are available when the API is first called.

Note: This special value only has meaning if '/' or '/asp_name' is specified for the *Path_Name* parameter (where asp_name is the name of an independent ASP which is available when the API is first called). Additionally, if *IN_EXclusion_ptr* is specified, it must only be as an exclusion array. If *ALLQSYS is specified, *NOQSYS cannot also be specified.«

Number of object types

The number of types included in the search.

Local_remote_obj

(Input) An unsigned integer that tells **Qp0lProcessSubtree()** whether to select only local objects, only remote objects, or both. Note that the decision of whether a file is local or remote varies according to the respective file system rules. Objects in file systems that do not carry either a local or remote indicator are treated as remote. Valid values follow:

0

QP0L_LOCAL_REMOTE_OBJ: Both local and remote objects are passed to the exit program.

1

QP0L_LOCAL_OBJ: Only local objects are passed to the exit program.

2

QP0L_REMOTE_OBJ: Only remote objects are passed to the exit program.

IN_EXclusion_ptr

(Input) A pointer to an array of pointers. Each pointer in the array points to a specific path name that identifies a directory, and all of its subdirectories, that **Qp0lProcessSubtree()** either includes or excludes in its search to find objects that meet the caller's input criteria. If this pointer is not NULL, the IN_EXclusion pointer type must indicate whether the list is an inclusive or exclusive list. The Number of pointers field must contain the number of path names for inclusion or exclusion on the search.

Use an inclusive list to specify multiple path names for searches on a single call to **Qp0lProcessSubtree()** versus using the *Path_Name* parameter, which searches only one path per call. The *Path_Name* parameter and an inclusive list are mutually exclusive. EINVAL is returned if both parameters are specified. The *IN_EXclusion_ptr* must be NULL if not used. All of the rules that apply to a single *Path_Name* entry apply to each inclusive list entry.

While an inclusion list allows the caller of **Qp0lProcessSubtree()** to identify multiple path names

for processing, **Qp0lProcessSubtree()** does not perform any verification to ensure uniqueness of path names or to verify any other relationship between path names entered in the inclusion array. For example, if the path names entered represent nested directories, **Qp0lProcessSubtree()** calls the exit program multiple times without any error message or other notification of this nesting.

Specify the root directory for a given file system as an exclusive list entry to eliminate that file system from a search.

All relative path names are relative to the current directory of the job that calls **Qp0lProcessSubtree()**.

The structure for this parameter follows.

IN_EXclusion Pointer.

This points to a list of path names to either include or exclude from a search.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	IN_EXclusion pointer type
4	4	BINARY(4)	Number of pointers
8	8	CHAR(8)	Reserved
16	10	ARRAY(*)	Path name pointers

IN_EXclusion pointer type

Whether a path name array contains directories that are included or contains directories that are excluded. Valid values follow:

0

QP0L_INCLUSION_TYPE: An inclusion array is identified.

1

QP0L_EXCLUSION_TYPE: An exclusion array is identified.

Number of pointers

The number of path name pointers that are in the inclusion or exclusion array.

Path name pointers

An array of pointers. Each pointer points to a path name that is included or excluded. Each path name must follow the Qlg_Path_Name_T structure. For more information on this structure, see Path Name Format.

Reserved

A reserved field. This field must be set to binary zero.

Err_recovery_action

(Input) An unsigned integer that describes how **Qp0lProcessSubtree()** handles errors that are not severe enough to force the API to end processing. Valid values follow:

0

QP0L_PASS_WITH_ERRORID: Calls the exit program and specifies the name (when

the name is available) of the object being accessed when an error occurs. This value also sends a valid *errno* to the exit program.

1

QP0L_BYPASS_NO_ERRORID: Bypasses the object being accessed when an error occurs, and moves to process the next object in the tree without notification to the calling program or to the exit program that an error has occurred.

2

QP0L_JOBLOG_NO_ERRORID: Sends message CPDA1C0 to the job log to identify the object being accessed when an error occurs. This value returns to process the next object without notification to the calling program or to the exit program that an error has occurred.

3

QP0L_NULLNAME_ERRORID: Calls the exit program with a NULL object name and a valid *errno*.

4

QP0L_END_PROCESS_SUBTREE: Quits **Qp0lProcessSubtree()** when an error occurs, and returns to the calling program, regardless of the error type. Note that the exit program is still given a call but cannot override the caller's decision to end processing. Calling the exit program allows the exit program to perform other tasks before the API returns to the caller. For example, the exit program can put information in the function control block that can be processed by the caller when the caller regains control.

UserFunction_ptr

(Input) A pointer to the name of an exit program that the caller wants **Qp0lProcessSubtree()** to call upon finding an object that matches the selection criteria. This exit program can be either a procedure or a program. See » Process a Path Name Exit Program « for the syntax of the user exit program.

The structure for this parameter follows.

User Function Pointer.

This points to the user exit program. The exit program can be a procedure or a program.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Function type flag
4	4	CHAR(10)	Program library
14	E	CHAR(10)	Program name
24	18	CHAR(1)	Multithreaded job action
25	19	CHAR(7)	Reserved
32	20	PP(*)	Procedure pointer to the exit procedure

Function type flag

An unsigned integer that indicates whether the user-supplied exit program that is called by

Qp0lProcessSubtree() is a procedure or a program. Valid values follow:

0

QP0L_USER_FUNCTION_PTR: A user procedure is called.

1

QP0L_USER_FUNCTION_PGM: A user program is called.

Multithreaded job action

(Input) A CHAR(1) value that indicates the action to take in a multithreaded job. The default value is QP0L_MLTTHDACN_SYSVAL. For release compatibility and for processing this parameter against the QMLTTHDACN system value, x'00, x'01', x'02', & x'03' are treated as x'F0', x'F1', x'F2', and x'F3'. Valid values follow:

x'00'

QP0L_MLTTHDACN_SYSVAL: The API evaluates the QMLTTHDACN system value to determine the action to take in a multithreaded job. Although the API can make repetitive calls to an exit program, the system value is evaluated once before Qp0lProcessSubtree() issues its first exit program call. This value is used on subsequent calls until the API returns control to its caller. Valid QMLTTHDACN system values follow:

'1'

Call the exit program. Do not send an informational message.

'2'

Call the exit program. Send informational message CPI3C80. Qp0lProcessSubtree() may call the exit program multiple times; however, this message is sent only once for each call to Qp0lProcessSubtree().

'3'

The exit program is not called when the API determines that it is running in a multithreaded job. ENOTSAFE is returned.

x'01'

QP0L_MLTTHDACN_NOMSG: Call the exit program. Do not send an informational message.

x'02'

QP0L_MLTTHDACN_MSG: Call the exit program. Send informational message CPI3C80. Qp0lProcessSubtree() may call the exit program multiple times; however, this message is sent only once for each call to Qp0lProcessSubtree().

x'03'

QP0L_MLTTHDACN_NO: The exit program is not called when the API determines that it is running in a multithreaded job. ENOTSAFE is returned.

Procedure pointer to the exit procedure

A procedure pointer to the procedure that **Qp0lProcessSubtree()** calls. This field must be NULL if a program is called instead of a procedure.

Program library

The library in which the called program, identified by Program name, is located. This field must be blank if a procedure is called instead of a program.

Program name

The name of the program that is called. The program is located in the library identified by Program library. This field must be blank if a procedure is called instead of a program.

Reserved

A reserved field. This field must be set to binary zero.

Function_CtlBlk_ptr

(Input) A pointer that **Qp0lProcessSubtree()** passes to the user-defined exit program that is called. **Qp0lProcessSubtree()** does not process this pointer or what is referred to by the pointer. It passes the pointer as a parameter to the user-defined exit program that was specified. This is a means for the caller of **Qp0lProcessSubtree()** to pass information to and from the Process a Path Name exit program.

Authorities

Note: Adopted authority is not used.

Authorization Required for

Qp0lProcessSubtree()

Object Referred to	Authority Required	errno
Each directory, preceding the last component, in a *Path Name*	*X	EACCES
The *Path Name* directory and all subdirectories of the *Path Name* that are included in the search.	*RX	EACCES
Each directory, preceding the last component, in any path name pointed to by the *IN_EXclusion ptr*	*X	EACCES
The *Path Name* directory and all subdirectories of any path name pointed to by an inclusive list	*RX	EACCES
Any called program pointed to by the *UserFunction_ptr* parameter	*X	EACCES
Any library that contains the called program pointed to by the *UserFunction_ptr* parameter	*X	EACCES

Return Value

0

Qp0lProcessSubtree() was successful.

-1

Qp0lProcessSubtree() was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **Qp0lProcessSubtree()** is not successful, the *errno* indicates one of the following errors:

[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.

[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.

[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.

[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.

[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.

[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.

[ENOTSAFE]

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

[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 message 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.
CPFA0D4 E	File system error occurred. Error number &1.
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 exit program called by **Qp0lProcessSubtree()** is not threadsafe or uses a function that is not threadsafe, then **Qp0lProcessSubtree()** is not threadsafe.
3. If the exit program called by **Qp0lProcessSubtree()** uses a function that fails when there are secondary threads active in the job, **Qp0lProcessSubtree()** may fail as a result.
4. Basic function and usage considerations
   - **Qp0lProcessSubtree()** does not perform the following tasks but is designed to work with the user exit function and other APIs to be useful in accomplishing the following and other tasks:
     - Retrieve object attributes (like authorities, dates, or sizes).
     - Build lists from selected objects.
     - Delete directories.
     - Identify multiple occurrences of an object within or across directories.
     - Count the number of objects in a directory.
   - **DosSetRelMaxFH()** is called to increase to the maximum the number of file descriptors that can be opened during processing such that **Qp0lProcessSubtree()** is not likely to fail due to a lack of descriptors. This value is not reset when **Qp0lProcessSubtree()** ends because the API could be running in a multithreaded job.
5. Object locking

   Qp0lProcessSubtree() does not perform any object locking, other than what is done when opening a directory to read the objects it contains, so that the exit program does not encounter or need to manage locks held by **Qp0lProcessSubtree()**. Once **Qp0lProcessSubtree()** has started searching a path, the addition, deletion, or removal of mounted directories or objects may not have any effect

on the results of the search.

If **Qp0lProcessSubtree()** encounters a directory that is locked, **Qp0lProcessSubtree()** uses the defined Err_recovery_action to determine how to handle the locked condition. Locks on objects that are not directories have no effect on **Qp0lProcessSubtree**().

6. Design considerations for parameters

    1. Symbolic links

        When the last component of the path name supplied on the initial call of **Qp0lProcessSubtree**() is a symbolic link, **Qp0lProcessSubtree()** resolves and follows the initial link to its target and performs its normal functions on the target. All other symbolic links that are encountered in the same search are not resolved to their targets.

        If the path name supplied on the initial call of **Qp0lProcessSubtree()** is a symbolic link that points to another file system or that points to a remote file system, the API resolves and processes the initial link only. It does not resolve other symbolic links that are encountered in the same search. However, if the caller specified that remote objects are not processed, but the initial path name (whether a symbolic link or not) points to a remote file system, the link is not resolved. **Qp0lProcessSubtree()** calls the exit program with a NULL path name and an indicator that **Qp0lProcessSubtree()** has completed successfully without any error indicators to the exit program.

        When *SYMLNK is specified as part of the selection criteria, **Qp0lProcessSubtree()** does not resolve the selected names.

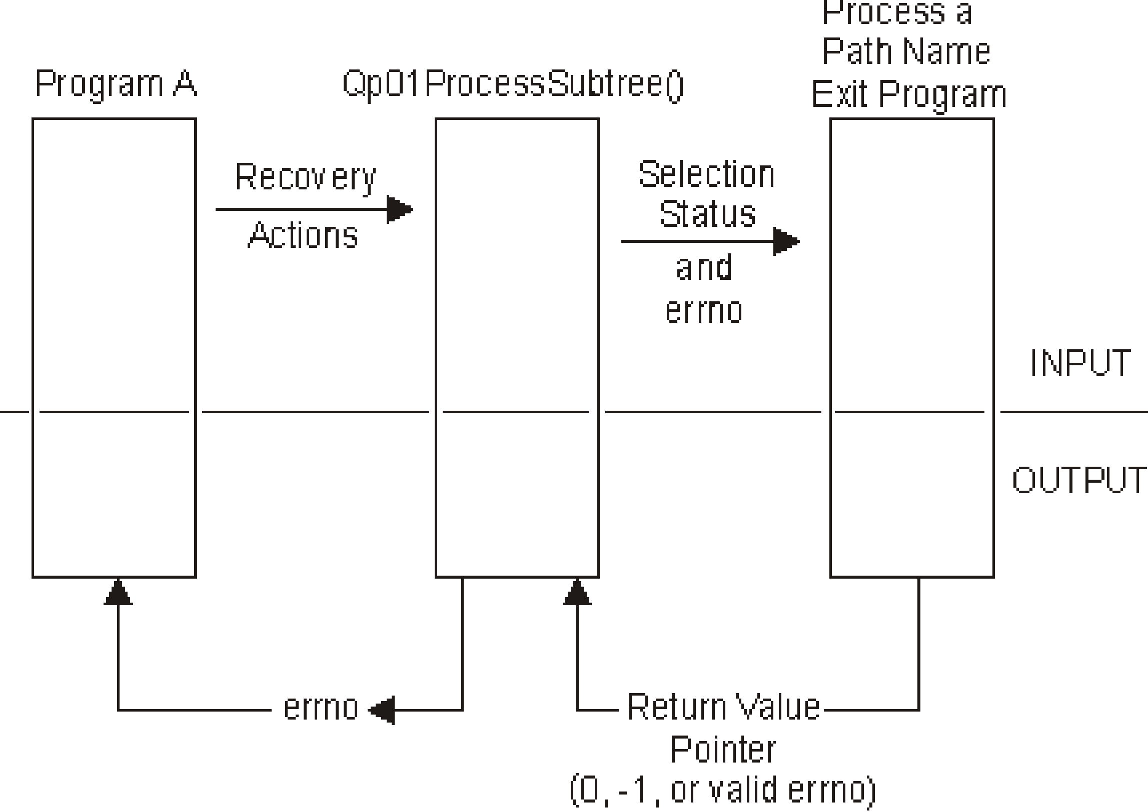
    2. Recovery Actions

        There are three separate parameters that control error recovery during a search. The caller of the API determines how an error should be reported to the exit program by setting the *Err_recovery_actions* parameter. The API sets the *Selection status pointer* and sends it to the exit program to indicate one of four conditions: the API search status is OK, the last object has been processed, the API has encountered recoverable errors, or the search cannot continue. For error conditions it also sends a valid *errno*. The exit program returns an indicator back to the API either to continue or to end the search by setting the *Return value pointer*. For error conditions, it also returns a valid *errno*, pointed to by the *Return value pointer*. Each time **Qp0lProcessSubtree()** regains control from the exit program, it determines whether the search should continue or end by evaluating the *Err_recovery_actions* parameter, its *Selection status pointer*, and the *Return value pointer*. Upon ending, **Qp0lProcessSubtree**() returns 0 to indicate a successful search, or a -1 and an *errno* to indicate the error condition. This *errno* may have been set by the exit program (*Return value pointer*).

        This error recovery design allows for flexibility in handling errors between the caller, the API, and the exit program. Whenever an unrecoverable error occurs, if possible, the exit program is given a final call; this call allows the exit program to do such tasks as cleanup or to put information in the function control block, or to record information about the error. However, the exit program cannot decide that the search should continue. The API will return to its caller when it regains control. There are only two specific instances in which the API determines that the exit program is not called:

        - When the API cannot resolve the exit program name or its authorization.
        - When input parameters are missing or specified incorrectly. (The API returns EINVAL to the caller before any other processing.)

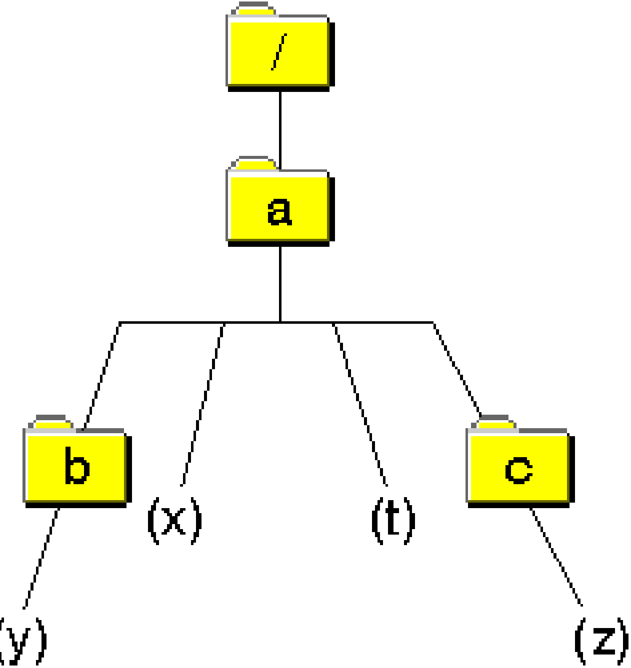
Following is a diagram showing the flow and relationship of these parameters.



Scenarios

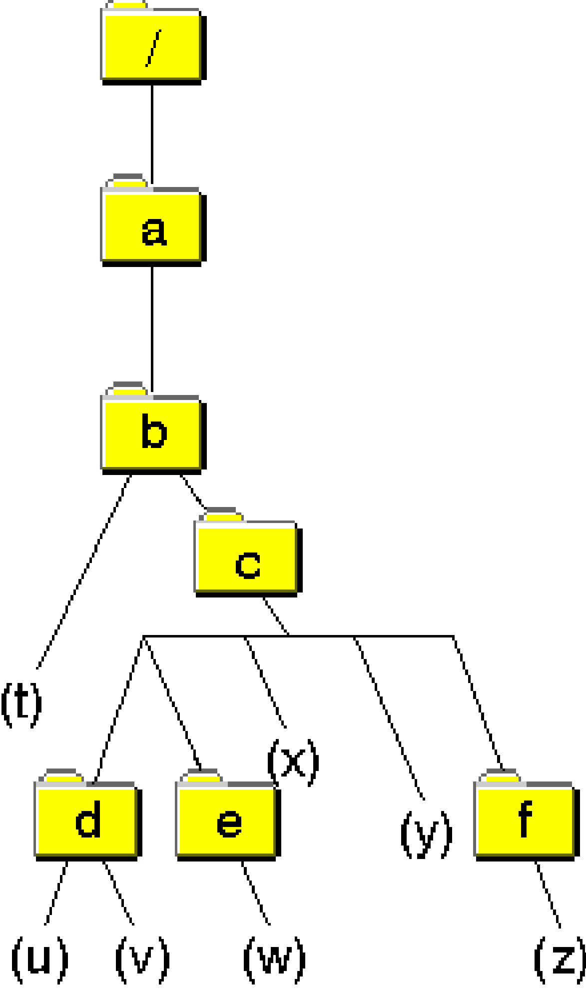
Following are scenarios showing calls and the results of calls to **Qp0lProcessSubtree()**. Directory Structure A and Directory Structure B define the input directory structure for these scenarios.

Figure: Directory Structure A



This directory structure represents three subdirectories (a, b, c), three objects (x, y, z), and a symbolic link (t).

Figure: Directory Structure B



This directory structure represents six subdirectories (a, b, c, d, e, f) and seven objects (t, u, v, w, x, y, z).

Scenario 1

This scenario assumes processing a directory as shown by *Directory Structure A* in Figure above.

This scenario shows a call to the API without any criteria to filter the selection of objects in the path being searched. If the API call were coded with the parameter values as shown by *Input value* in Scenario 1 API Input, the exit program would be called nine times and would pass the object names as shown by the *Object Name Pointer* in Results of a call. Because QP0L_SUBTREE_YES is specified, all of the directories in the path will be opened and the name of all the objects that they contain will be passed to the exit program. Note that the only guaranteed order is that parent directories are passed to the exit program after all of their children.

Figure: Scenario 1 API Input

Input Parameter	Input value
*Path_Name	'/' ('/' processes every directory on the system and is not recommended if performance is a consideration)

Subtree_level	QP0L_SUBTREE_YES
*Objtypes_array_ptr	NULL
Local_remote_obj	QP0L_LOCAL_REMOTE_OBJ
*IN_EXclusion_ptr	NULL
Err_recovery_action	QP0L_PASS_WITH_ERRORID
*UserFunction_ptr	QP0L_USER_FUNCTION_PTR
*Function_CtlBlk_ptr	NULL

Figure: Results of a call

Exit Program Call Count	Object Name Pointer
1	/a/b/y
2	/a/b
3	/a/x
4	/a/t
5	/a/c/z
6	/a/c
7	/a
8	/
9	NULL path name (indicates the API completed)

Scenario 2

This scenario assumes processing a directory as shown by *Directory Structure A* in the Figure above.

This shows a call to the API with the *Subtree level* parameter set to retrieve only one level, without any object filtering. Since QP0L_SUBTREE_NO is specified, the names of all objects in the path will be passed to the exit program, however, none of the directories will be opened. This allows a caller to perform tasks such as identifying all of the root objects for a file system. For example, this would identify all of the first level folders, when processing against the QDLS file system. Then the API can be called recursively from within the exit program, with each of these folders specified as the path to be searched.

If the API call were coded with the parameter values as shown by *Input value* in Scenario 2 API Input, the exit program would be called six times and would pass the object names as shown by the *Object Name*

Pointer in Results of a call.

Figure: Scenario 2 API Input

Input Parameter	Input value
*Path_Name	'/a'
Subtree_level	QP0L_SUBTREE_NO
*Objtypes_array_ptr	NULL
Local_remote_obj	QP0L_LOCAL_REMOTE_OBJ
*IN_EXclusion_ptr	NULL
Err_recovery_action	QP0L_PASS_WITH_ERRORID
*UserFunction_ptr	QP0L_USER_FUNCTION_PTR
*Function_CtlBlk_ptr	NULL

Figure: Results of a call

Exit Program Call Count	Object Name Pointer
1	/a/b
2	/a/x
3	/a/t
4	/a/c
5	/a
6	NULL path name (indicates the API completed)

Scenario 3

This scenario assumes processing a directory as shown by *Directory Structure B* in the Figure above.

This scenario represents a call to the API with an inclusion list. Note that the *Path Name* parameter is not used as the starting directory since each entry in an inclusion list is treated as a starting directory.

If the API call were coded with the parameter values as shown by *Input value* in Scenario 3 API Input, the exit program would be called six times and would pass the object names as shown by the *Object Name*

Pointer in Results of a call.

Note that /a/b/c/d/v could be returned before /a/b/c/d/u, as shown in this scenario, since children in a directory can be returned in any order. The only guaranteed order is that the exit program is called with all children objects before being called with the parent to allow the exit program to delete directories if desired.

Figure: Scenario 3 API Input

Input Parameter	Input value
*Path_Name	NULL (not used with an inclusion list)
Subtree_level	QP0L_SUBTREE_YES
*Objtypes_array_ptr	'*DIR ' '*STMF '
Local_remote_obj	QP0L_LOCAL_OBJ
*IN_EXclusion_ptr	QP0L_INCLUSION_TYPE, '/a/b/c/d/' '/a/b/c/e/'
Err_recovery_action	QP0L_PASS_WITH_ERRORID
*UserFunction_ptr	QP0L_USER_FUNCTION_PTR
*Function_CtlBlk_ptr	NULL

Figure: Results of a call

Exit Program Call Count	Object Name Pointer
1	/a/b/c/d/v
2	/a/b/c/d/u
3	/a/b/c/d
4	/a/b/c/e/w
5	/a/b/c/e/
6	NULL path name (indicates the API completed)

Scenario 4

This scenario assumes processing a directory as shown by *Directory Structure B* in the Figure above.

This scenario represents a call to the API with an exclusion list. Note that each relative entry in the exclusion list is resolved relative to the current working directory at the time the API is called. This scenario assumes that the current working directory is /a/b/.

If the API call were coded with the parameter values as shown by *Input value* in Scenario 4 API Input, the exit program would be called eight times and would pass the object names as shown by the *Object Name Pointer* in Results of a call.

This scenario also shows that children in a directory can be returned in any order. The only guaranteed order is that the exit program is called with all children objects before being called with the parent to allow the exit program to delete directories if desired.

Figure: Scenario 4 API Input

Input Parameter	Input value
*Path_Name	'/a/b/'
Subtree_level	QP0L_SUBTREE_YES
*Objtypes_array_ptr	'*DIR ' '*STMF '
Local_remote_obj	QP0L_LOCAL_OBJ
*IN_EXclusion_ptr	QP0L_EXCLUSION_TYPE, 'c/d/' 'c/e/'
Err_recovery_action	QP0L_PASS_WITH_ERRORID
*UserFunction_ptr	QP0L_USER_FUNCTION_PTR
*Function_CtlBlk_ptr	NULL

Figure: Results of a call

Exit Program Call Count	Object Name Pointer
1	/a/b/t
2	/a/b/c/y
3	/a/b/c/f/z
4	/a/b/c/f
5	/a/b/c/x
6	/a/b/c
7	/a/b

8	NULL path name (indicates the API completed)

Related Information

- The <**Qp0lstdi.h**> file (see Header Files for UNIX-Type Functions)
- The <**qlg.h**> file (see Header Files for UNIX-Type Functions)
- QlgProcessSubtree()--Process a Path Name (using NLS-enabled path name)
- Process a Path Name Exit Program

Example

See Code disclaimer information for information pertaining to code examples.

Following is a code example showing a call to the Qp0lProcessSubtree() API with a procedure as the exit program:

```
/*****************************************************************/
/*****************************************************************/

#include <Qp0lstdi.h>
#include <stdio.h>
#include <errno.h>
#include <qtqiconv.h>

void Obj_Print_Function
                    (uint            *Selection_status_pointer,
                     uint            *Error_value_pointer,
                     uint            *Return_value_pointer,
                     Qlg_Path_Name_T *Object_name_pointer,
                     void            *Function_control_block_pointer)
{
 /****************************************************************/
 /* This exit program example prints the names, one at a time,  */
 /* of each entry in a directory structure that it receives on  */
 /* each call from Qp0lProcessSubtree().                        */
 /****************************************************************/

 #define PATH_TYPE_POINTER     0x00000001  /* If this flag is on, */
                             /* the qlg structure contains a    */
                             /* pointer to the path name.       */
                             /* Otherwise, the path name is in  */
                             /* contiguous storage within the   */
                             /* qlg structure.                  */
```

```
 typedef union pn_input_type
 {
     char pn_char_type[256];     /* path name is in    */
                                          /* contiguous storage */
     char *pn_ptr_type;       /* path name is a pointer          */
 };
typedef struct pnstruct
    {
     Qlg_Path_Name_T qlg_struct;
     union pn_input_type   pn;
    };
 struct pnstruct *pns;
 char *path_ptr;

 size_t insz;
 size_t outsz = 1000;
 char outbuf[1000];
 char *outbuf_ptr;
 iconv_t cd;
 size_t ret_iconv;

 QtqCode_T toCode   =    {37,0,0,0,0,0};
 QtqCode_T fromCode = {61952,0,0,1,0,0};

 if (*Selection_status_pointer == QP0L_SELECT_OK)
  {
   if (Object_name_pointer != NULL)
    {
     /**********************************************************/
     /* Point to the pathname and get the size of the pathname   */
     /* that was sent from the Qp0lProcessSubtree() API.  The    */
     /* format of the pathname must be determined by evaluating  */
     /* Path_Type in the qlg structure.                          */
     /**********************************************************/
     pns = (struct pnstruct *)Object_name_pointer;
     if (Object_name_pointer->Path_Type & PATH_TYPE_POINTER)
     {
  path_ptr = pns->pn.pn_ptr_type;
     }
     else
     {
       path_ptr = (char *)(pns->pn.pn_char_type);
     }
     insz = pns->qlg_struct.Path_Length;


     /**********************************************************/
     /* Initialize the print buffer.                             */
     /**********************************************************/
     outbuf_ptr = (char *)outbuf;
     memset(outbuf_ptr, 0x00, insz);


     /**********************************************************/
     /* Use iconv to convert from 61952 to the job CCSID.        */
     /* REMEMBER iconv will change the data that it receives.    */
     /**********************************************************/
```

```
      cd = /* Open the conversion descriptor.*/
           QtqIconvOpen(&toCode,
                        &fromCode);
      if (cd.return_value == -1)
        {
         /*********************************************************/
         /* If conversion descriptor was not opened successfully, */
         /* return an error and errno (ECONVERT) to the API.      */
         /*********************************************************/
         *Return_value_pointer = errno;
         return;
         }

      ret_iconv = /* Perform the conversion.*/
                  (iconv(cd,
                        (char **)&(path_ptr),
                        &insz,
                        (char **)&(outbuf_ptr),
                        &outsz));
      if (ret_iconv != 0)
        {
         /*********************************************************/
         /* If the conversion failed, close the conversion        */
         /* descriptor and return an error and errno (ECONVERT)   */
         /* to the API.                                           */
         /*********************************************************/
         ret_iconv= iconv_close(cd);
         *Return_value_pointer = errno;
         return;
         }

      /************************************************************/
      /* Print the name of the object being processed and close   */
      /* the conversion descriptor.                               */
      /************************************************************/
      printf("In User Exit Program. Path is %s.\n", outbuf);
      ret_iconv = iconv_close(cd);

    }                         /* end Object_name_pointer != NULL */
    else
       {
        printf"In User Exit Program with a null Pathname \n");
       }
  }               /* end *Selection_status_pointer == QP0L_SELECT_OK */

      *Return_value_pointer = 0;

} /* end Exit program */



int main (int argc, char *argv[])
  {
   #define MYPN "/TestDir"
   const int  zero = 0;
   const char US_const[3]= "US";
   const char Language_const[4]="ENU";
```

```
  const char Path_Name_Del_const[2]= "/";
  const char LibObj_const[12]= "*LIB       ";
  typedef struct pnstruct
   {
    Qlg_Path_Name_T  qlg_struct;
    char             pn[50];  /* Must be greater than  */
                                        /* or equal the length   */
                                        /* of the path name.     */
    };
  struct pnstruct pns;
  Qp0l_Objtypes_List_t   MyObj_types;
  Qp0l_User_Function_t   User_function;
  struct
    {
     uint    AnyData_to_the_exitprogram;
     uint    AnyData_not_processed_by_the_API;
    } CtlBlkAreaName;

  int rc;
 /*****************************************************************/
 /* In this example, the pathname is defined by MYPN as TestDir */
 /* and it is assumed that the TestDir directory exists on the  */
 /* system.  Various other functions or other routines could be */
 /* included here to (for example):                             */
 /*    1) determine the beginning search directory.             */
 /*    2) construct the path name in the correct format.        */
 /*    3) others...                                             */
 /*****************************************************************/

 /*****************************************************************/
 /*****************************************************************/
 /*   Initialize Qp0lProcessSubtree() API Parameters            */
 /*****************************************************************/
 memset((void*)&pns, 0x00, sizeof(struct pnstruct));
 pns.qlg_struct.CCSID = 37;
 memcpy(pns.qlg_struct.Country_ID,US_const,2);
 memcpy(pns.qlg_struct.Language_ID,Language_const,3);
 pns.qlg_struct.Path_Type = zero;
 pns.qlg_struct.Path_Length = sizeof(MYPN)-1;
 memcpy(pns.qlg_struct.Path_Name_Delimiter,Path_Name_Del_const,1);
 memcpy(pns.pn,MYPN,sizeof(MYPN));
 MyObj_types.Number_Of_Objtypes = zero;
 memset((void *)&User_function, 0x00, sizeof(Qp0l_User_Function_t));
 User_function.Function_Type = QP0L_USER_FUNCTION_PTR;
 User_function.Mltthdacn[0] = QP0L_MLTTHDACN_NOMSG;
 User_function.Procedure = &Obj_Print_Function;

if (rc = Qp0lProcessSubtree((Qlg_Path_Name_T *)&pns,
                            QP0L_SUBTREE_YES,
                            (Qp0l_Objtypes_List_t *)NULL,
                            QP0L_LOCAL_REMOTE_OBJ,
                            (Qp0l_IN_EXclusion_List_t *)NULL,
                            QP0L_PASS_WITH_ERRORID,
                            &User_function,
                            &CtlBlkAreaName) == 0)
  {
    printf("Qp0lProcessSubtree() Successful : error = %d\n", errno);
```

```
   }
 else
     {/*unsuccessful return from Qp0lProcessSubtree() API   */
      printf("ERROR on Qp0lProcessSubtree(): error = %d\n", errno);
      perror("Error message");
     }
  }        /* end main */
```

API introduced: V4R3

Qp0lRenameKeep()--Rename File or Directory, Keep "new" If It Exists

Syntax

```
#include <Qp0lstdi.h>

int Qp0lRenameKeep(const char *old, const char *new);
```

Threadsafe: Conditional; see Usage Notes.

The **Qp0lRenameKeep()** function renames a file or a directory specified by *old* to the name given by *new*. The *old* pointer must specify the name of an existing file or directory. Both *old* and *new* must be of the same type; that is, both directories or both files. *old* and *new* must not end in "dot" (.) or "dot-dot" (..).

If *new* already exists, **Qp0lRenameKeep()** fails with the [EEXIST] error.

If the *old* argument points to a symbolic link, the symbolic link is renamed. **Qp0lRenameKeep()** does not affect any file or directory named by the contents of the symbolic link. See Usage Notes for more information.

When **Qp0lRenameKeep()** is successful, it updates the change and modification times for the parent directories of *old* and *new*.

If the *old* object is checked out, **Qp0lRenameKeep()** fails with the [EBUSY] error.

Parameters

old

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

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 QlgRenameKeep()--Rename File or Directory, Keep "new" If It Exists (using NLS-enabled path name) for a description and an example of supplying the *old* in any CCSID.

new

(Input) A pointer to the null-terminated path name of the new name of the file.

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 file name is assumed to be represented in the language and country or region currently in effect for the job.

See QlgRenameKeep()--Rename File or Directory, Keep "new" If It Exists (using NLS-enabled path name) for a description and an example of supplying the *new* in any CCSID.

Authorities

Note: Adopted authority is not used.

Figure 1-57. Authorization Required for Qp0lRenameKeep() (excluding QSYS.LIB, independent ASP QSYS.LIB, QDLS, and QOPT)

Object Referred to	Authority Required	errno
Each directory in *old* path name preceding the object to be renamed	*X	EACCES
Parent directory of *old* object	*WX	EACCES
old object if it is a directory	*OBJMGT + *W	EACCES
old object if it is not a directory	*OBJMGT	EACCES
Each directory in *new* path name preceding the object	*X	EACCES
Parent directory of *new* object	*WX	EACCES

Figure 1-58. Authorization Required for Qp0lRenameKeep() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Each directory in *old* path name preceding the object to be renamed	*X	EACCES
Parent directory of *old* object if the object is a database file member	*OBJMGT	EACCES
Parent directory of the parent directory of *old* object if the object is a database file member	*UPD	EACCES
Parent directory of *old* object if the object is not a database file member	See the QLIRNMO API for details	EACCES
old object if it is a database file member	None	None
old object if it is not a database file member	See the QLIRNMO API for details	EACCES
Each directory in *new* path name preceding the object	*X	EACCES
Parent directory of *new* object	See the QLIRNMO API for details	EACCES

Figure 1-59. Authorization Required for Qp0lRenameKeep() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in *old* path name preceding the object to be renamed	*X	EACCES
Parent directory of *old* object	*CHANGE	EACCES
old object	*ALL	EACCES
Each directory in *new* path name preceding the object	*X	EACCES
Parent directory of *new* object	*CHANGE	EACCES

Figure 1-60. Authorization Required for Qp0lRenameKeep() in the QOPT File System

Object Referred to	Authority Required	errno
Volume authorization list for volume to be renamed in a media library device	*ALL	EACCES
Volume authorization list for volume to be renamed in a stand alone device	*CHANGE	EACCES
Volume authorization list for volume containing object to be renamed	*CHANGE	EACCES
Root directory (/) of volume to be renamed if volume media format is Universal Disk Format (UDF)	*RWX	EACCES
Each directory in old path name preceding the object to be renamed if volume media format is Universal Disk Format (UDF)	*X	EACCES
Parent directory of old object if volume media format is Universal Disk Format (UDF)	*WX	EACCES
Old object if volume media format is Universal Disk Format (UDF)	*W	EACCES
Each directory in new path name preceding the object if volume media format is Universal Disk Format (UDF)	*X	EACCES
Parent directory of new object if volume media format is Universal Disk format (UDF)	*WX	EACCES
Object and parent directories if volume media format is not Universal Disk format (UDF)	None	None

Return Value

0

Qp0lRenameKeep() was successful.

-1

Qp0lRenameKeep() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **Qp0lRenameKeep()** 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. May be returned if the directories preceding the object to be renamed in the *old* path name are part of *new*, or if either name refers to dot or dot-dot.

[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. New is a directory, but old is not 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.

[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.

[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.

[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.

old is a directory and the link count of the parent directory of *new* would exceed LINK_MAX.

[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.

[EXDEV]

Improper link.

A link to a file on another file system was attempted.

old and *new* identify files or directories in different file systems. Links between different file systems are not allowed.

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. About the Rename Functions

   The integrated file system provides two functions that rename a file or directory. Both rename the *old* path name to a *new* path name. The difference is determined by what happens when *new* already exists:

- If *new* already exists when using **Qp0lRenameKeep()**, the rename fails with the [EEXIST] error.
- If *new* already exists when using **Qp0lRenameUnlink()**, the existing path name is unlinked (removed) before *old* is renamed to *new*.

These functions are defined in the **<Qp0lstdi.h>** header file. When **<Qp0lstdi.h>** is included, the **rename()** function is defined to be either **Qp0lRenameUnlink()** or **Qp0lRenameKeep()**, depending on the definitions of the _POSIX_SOURCE and _POSIX1_SOURCE macros:

- When _POSIX_SOURCE and _POSIX1_SOURCE are **not** defined, **rename()** is defined to be **Qp0lRenameKeep()**. Either **rename()** or **Qp0lRenameKeep()** can be used to rename a file or directory with the semantics of **Qp0lRenameKeep()**.
- When _POSIX_SOURCE or _POSIX1_SOURCE is defined, **rename()** is defined to be **Qp0lRenameUnlink()**. Either **rename()** or **Qp0lRenameUnlink()** can be used to rename a file or directory with the semantics of **Qp0lRenameUnlink()**.

When the **<Qp0lstdi.h>** header file is **not** included, **rename()** operates only on database files in the QSYS.LIB and independent ASP QSYS.LIB file systems, as it did before the introduction of the integrated file system.

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

   - When a database member is being renamed, the part of the *new* path name preceding the object must be the same as that of the *old* path name. That is, the sequence of "directories" (library and file) preceding the object in the *new* path name must be the same as the sequence of directories preceding the object in the *old* path name.
   - The following object types cannot be renamed when there are secondary threads active in the job: *CFGL, *CNNL, *CTLD, *DEVD, *LIND, *NWID. The operation will fail with error code [ENOTSAFE].
   - When a library is being renamed, the part of the *new* path name preceding the object must be the same as that of the *old* path name. That is, the sequence of "directories" (/QSYS.LIB or /asp_name/QSYS.LIB, where asp_name is the independent Auxiliary Storage Pool name) preceding the object in the *new* path name must be the same as the sequence of directories preceding the object in the *old* path name [EINVAL].

4. QDLS File System Differences

   When a folder is being renamed, the part of the *new* path name preceding the object must be the same as that of the *old* path name. That is, a folder must remain in the same parent folder.

5. QOPT File System Differences

   You can rename only a volume or a file, not a directory.

6. QFileSvr.400 File System Differences

   You cannot rename the first-level directory. For example, you cannot rename Dir1 in the path name /QFileSvr.400/Dir1/Dir2/Object. The first-level directory identifies the target system in a communications connection.

7. QNetWare File System Differences

The new and old files or directories must exist on the same NetWare server. This function cannot be used to move data from one server to another.

8. QNTC File System Differences

   The *new* and the *old* files or directories must exist on the same Windows NT server. This function cannot be used to move data from one server to another.

9. Root (/) and User-defined File System Differences

   If the file being renamed is in the root file system or in a monocase user-defined file system, and the file system has the *TYPE2 directory format, and both *old* and *new* refer to the same link, but their case is different (eg. /ABC and /Abc), Qp0lRenameUnlink() changes the link name to the *new* name.

Related Information

- The <**stdio.h**> file (see Header Files for UNIX-Type Functions)
- The <**Qp0lstdi.h**> file (see Header Files for UNIX-Type Functions)
- pathconf()--Get Configurable Path Name Variables
- rename()--Rename File or Directory
- QlgRenameKeep()--Rename File or Directory, Keep "new" If It Exists (using NLS-enabled path name)
- Qp0lRenameUnlink()--Rename File or Directory, Unlink "new" If It Exists

Example

When you pass two file names to this example, it will try to change the file name from the first name to the second using **Qp0lRenameKeep()**.

```
#include <Qp0lstdi.h>

int main(int argc, char ** argv ) {

  if ( argc != 3 )
    printf( "Usage: %s old_fn new_fn\n", argv[0] );
  else if ( Qp0lRenameKeep( argv[1], argv[2] ) != 0
    perror ( "Could not rename file" );
}
```

API introduced: V3R1

Qp0lRenameUnlink()--Rename File or Directory, Unlink "new" If It Exists

Syntax

```
#include <Qp0lstdi.h>

int Qp0lRenameUnlink(const char *old, const char *new);
```

Threadsafe: Conditional; see Usage Notes.

The **Qp0lRenameUnlink()** function renames a file or a directory specified by *old* to the name given by *new*. The *old* pointer must specify the name of an existing file or directory. Both *old* and *new* must be of the same type; that is, both directories or both files. *old* and *new* must not end in "dot" (.) or "dot-dot" (..).

If *new* already exists, it is removed before *old* is renamed to *new*. Therefore, if *new* specifies the name of an existing directory, the directory must be empty.

If the *old* argument points to a symbolic link, the symbolic link is renamed. If the *new* argument points to a symbolic link, the link is removed and *old* is renamed to *new*. **Qp0lRenameUnlink()** does not affect any file or directory named by the contents of the symbolic link.

If *old* and *new* both refer to the same file, **Qp0lRenameUnlink()** returns successfully and performs no other action. See Usage Notes for more information.

When **Qp0lRenameUnlink()** is successful, it updates the change and modification times for the parent directories of *old* and *new*.

If the *old* object is checked out, **Qp0lRenameUnlink()** fails with the [EBUSY] error.

Parameters

old

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

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 QlgRenameUnlink()--Rename File or Directory, Unlink "new" If It Exists (using NLS-enabled path name) for a description and an example of supplying the *old* in any CCSID.

new

(Input) A pointer to the null-terminated path name of the new name of the file.

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 file name is assumed to be represented in the language and country or region currently in effect for the process.

See QlgRenameUnlink()--Rename File or Directory, Unlink "new" If It Exists (using NLS-enabled path name) for a description and an example of supplying the *new* in any CCSID.

Authorities

Note: Adopted authority is not used.

Figure 1-61. Authorization Required for Qp0lRenameUnlink() (excluding QSYS.LIB, independent ASP QSYS.LIB, QDLS, and QOPT)

Object Referred to	Authority Required	errno
Each directory in *old* path name preceding the object to be renamed	*X	EACCES
Parent directory of *old* object	*WX	EACCES
old object if it is a directory	*OBJMGT + *W	EACCES
old object if it is not a directory	*OBJMGT	EACCES
Each directory in *new* path name preceding the object	*X	EACCES
Parent directory of *new* object	*WX	EACCES
New object, if it exists	*OBJEXIST	EACCES

Figure 1-62. Authorization Required for Qp0lRenameUnlink() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Each directory in *old* path name preceding the object to be renamed	*X	EACCES
Parent directory of *old* object if the object is a database file member	*OBJMGT	EACCES
Parent directory of the parent directory of *old* object if the object is a database file member	*UPD	EACCES
Parent directory of *old* object if the object is not a database file member	See the QLIRNMO API for details	EACCES
old object if it is a database file member	None	None
old object if it is not a database file member	See the QLIRNMO API for details	EACCES
Each directory in *new* path name preceding the object	*X	EACCES
Parent directory of *new* object	See the QLIRNMO API for details	EACCES

Figure 1-63. Authorization Required for Qp0lRenameUnlink() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in *old* path name preceding the object to be renamed	*X	EACCES
Parent directory of *old* object	*CHANGE	EACCES
old object	*ALL	EACCES
Each directory in *new* path name preceding the object	*X	EACCES
Parent directory of *new* object	*CHANGE	EACCES

Figure 1-64. Authorization Required for Qp0lRenameUnlink() in the QOPT File System

Object Referred to	Authority Required	errno
Volume to be renamed	*ALL	EACCES
Volume containing object to be renamed	*CHANGE	EACCES
Object within volume	None	None

Return Value

0

Qp0lRenameUnlink() was successful.

-1

Qp0lRenameUnlink() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **Qp0lRenameUnlink()** 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. May be returned if the directories preceding the object to be renamed in the *old* path name are part of *new*, or if either name refers to dot or dot-dot.

[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. New is a directory, but old is not 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.

[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.

[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.

[ENOTEMPTY]

Directory not empty.

You tried to remove a directory that is not empty. A directory cannot contain objects when it is being removed.

The specified directory is not empty.

[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.

[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.

[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.

old is a directory and the link count of the parent directory of *new* would exceed LINK_MAX.

[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.

[EXDEV]

Improper link.

A link to a file on another file system was attempted.

old and *new* identify files or directories on different file systems. Links between different file systems are not allowed.

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. About the Rename Functions

   The integrated file system provides two functions that rename a file or directory. Both rename the *old* path name to a *new* path name. The difference is determined by what happens when *new* already exists:
   - If *new* already exists when using **Qp0lRenameUnlink()**, the existing path name is unlinked (removed) before *old* is renamed to *new*.

- If *new* already exists when using **Qp0lRenameKeep()**, the rename fails with the [EEXIST] error.

These functions are defined in the **<Qp0lstdi.h>** header file. When **<Qp0lstdi.h>** is included, the **rename()** function is defined to be either **Qp0lRenameUnlink()** or **Qp0lRenameKeep()**, depending on the definitions of the _POSIX_SOURCE and _POSIX1_SOURCE macros:

- When _POSIX_SOURCE or _POSIX1_SOURCE is defined, **rename()** is defined to be **Qp0lRenameUnlink()**. Either **rename()** or **Qp0lRenameUnlink()** can be used to rename a file or directory with the semantics of **Qp0lRenameUnlink()**.
- When _POSIX_SOURCE and _POSIX1_SOURCE are **not** defined, **rename()** is defined to be **Qp0lRenameKeep()**. Either **rename()** or **Qp0lRenameKeep()** can be used to rename a file or directory with the semantics of **Qp0lRenameKeep()**.

When the **<Qp0lstdi.h>** header file is **not** included, **rename()** operates only on database files in the QSYS.LIB and » independent ASP QSYS.LIB file systems, «as it did before the introduction of the integrated file system.

3. QSYS.LIB and »Independent ASP QSYS.LIB «File System Differences
   - When a database member is being renamed, the part of the *new* path name preceding the object must be the same as that of the *old* path name. That is, the sequence of "directories" (library and file) preceding the object in the *new* path name must be the same as the sequence of directories preceding the object in the *old* path name. If *new* already exists, »[EEXIST] is returned.«
   - The following object types cannot be renamed when there are secondary threads active in the job: *CFGL, *CNNL, *CTLD, *DEVD, *LIND, *NWID. The operation will fail with error code [ENOTSAFE].
   - »When a library is being renamed, the part of the *new* path name preceding the object must be the same as that of the *old* path name. That is, the sequence of "directories" (/QSYS.LIB or /asp_name/QSYS.LIB, where asp_name is the independent Auxiliary Storage Pool name) preceding the object in the *new* path name must be the same as the sequence of directories preceding the object in the *old* path name. «

4. QDLS File System Differences

   When a folder is being renamed, the part of the *new* path name preceding the object must be the same as that of the *old* path name. That is, a folder must remain in the same parent folder.

   If the object identified by the *new* path name exists, QDLS returns the [EEXIST] error.

5. QOPT File System Differences

   You can rename only a volume or a file, not a directory.

   If the object identified by the new path name exists, QOPT returns the [EEXIST] error.

6. QFileSvr.400 File System Differences

   You cannot rename the first-level directory. For example, you cannot rename Dir1 in the path name /QFileSvr.400/Dir1/Dir2/Object. The first-level directory identifies the target system in a communications connection.

7. QNetWare File System Differences

   The new and old files or directories must exist on the same NetWare server. This function cannot be used to move data from one server to another.

8. QNTC File System Differences

   The *new* and the *old* files or directories must exist on the same Windows NT server. This function cannot be used to move data from one server to another.

9. Root (/) and User-defined File System Differences

   If the file being renamed is in the root file system or in a monocase user-defined file system, and the file system has the *TYPE2 directory format, and both *old* and *new* refer to the same link, but their case is different (eg. /ABC and /Abc), Qp0lRenameUnlink() changes the link name to the *new* name.

Related Information

- The <**stdio.h**> file (see Header Files for UNIX-Type Functions)
- The <**Qp0lstdi.h**> file (see Header Files for UNIX-Type Functions)
- pathconf()--Get Configurable Path Name Variables
- rename()--Rename File or Directory
- Qp0lRenameKeep()--Rename File or Directory, Keep "new" If It Exists
- QlgRenameUnlink()--Rename File or Directory, Unlink "new" If It Exists (using NLS-enabled path name)

Example

When you pass two file names to this example, it will try to change the file name from the first name to the second using **Qp0lRenameUnlink()**.

```
#include <Qp0lstdi.h>

int main(int argc, char ** argv ) {

  if ( argc != 3 )
    printf( "Usage: %s old_fn new_fn\n", argv[0] );
  else if ( Qp0lRenameUnlink( argv[1], argv[2] ) != 0
    perror ( "Could not rename file" );
}
```

API introduced: V3R1

»Retrieve Object References (QP0LROR) API

Syntax

```
#include <qp0lror.h>
void QP0LROR(
             void *              Receiver_Ptr,
             unsigned int        Receiver_Length,
             char *              Format_Ptr,
             Qlg_Path_Name_T *   Path_Ptr,
             void *              Error_Code_Ptr
            );
```

Default Public Authority: *USE

Threadsafe: Yes

The **QP0LROR()** API is used to retrieve information about Integrated File System references on an object.

A reference is an individual type of access or lock obtained on the object when using Integrated File System interfaces. An object may have multiple references concurrently held, provided that the reference types do not conflict with one another.

This API will not return information about byte range locks that may currently be held on an object.

Parameters

Receiver_Ptr

(Output)

The variable that is to receive the information requested. You can specify the size of this area to be smaller than the format requested as long as you specify the length parameter correctly. As a result, the API returns only the data that the area can hold.

The format of the output is described by either the RORO0100 output format or the RORO0200 output format. See RORO0100 Output Format Description or the RORO0200 Output Format Description for a detailed description of these output formats.

Receiver_Length

(Input)

The length of the receiver variable. If the length is larger than the size of the receiver variable, the results may not be predictable. The minimum length is 8 bytes.

Format_Ptr

(Input)

Pointer to an 8-byte character string that identifies the desired output format. It must be one of the following values:

RORO0100 The reference type output will be formatted in a RORO0100 format. See RORO0100 Output Format Description. This format gives the caller a quick view of the object's references.

RORO0200 The reference type output will be formatted in a RORO0200 format. See RORO0200 Output Format Description. Specifying this format may cause QP0LROR to be a long running operation. The length of time it will take to complete depends on the number of jobs active on the system, and the number of jobs currently using objects through Integrated File System interfaces.

Path_Ptr

(Input)

Pointer to the path name to the object whose reference information is to be obtained. The path name must be specified in an NLS-enabled format specified by the Qlg_Path_Name structure. For more information on the Qlg_Path_Name_T structure, see Path name format.

If the last element of the *path* is a symbolic link, the **Qp0lROR()** function does not resolve the contents of the symbolic link. The reference information will be obtained for the symbolic link itself.

Error_Code_Ptr

(Input/Output)

Pointer to an error code structure to receive error information. See Error code parameter for more information.

Authorities and Locks

Directory Authority

The user must have execute (*X) data authority to each directory preceding the object whose references are to be obtained.

Object Authority

The user must have read (*R) data authority to the object whose references are to be obtained.

Output Structure Formats

RORO0100 Output Format Description (*Qp0l_RORO0100_Output*)

This structure is used to return object reference information.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4), UNSIGNED	Bytes returned
4	4	BINARY(4), UNSIGNED	Bytes available
8	8	BINARY(4), UNSIGNED	Offset to simple reference types

12	0C	BINARY(4), UNSIGNED	Length of simple reference types
16	10	BINARY(4), UNSIGNED	Reference count
20	14	BINARY(4), UNSIGNED	In-use indicator
Offset determined from *Offset to Simple Reference Types* field		Qp0l_Sim_Ref_Types_Output Structure	Simple reference types structure. See Simple object reference types structure description for a description of this structure.

RORO0200 Output Format Description (*Qp0l_RORO0200_Output*)

This output format is used to return object reference information, including a list of jobs known to be referencing the object. This includes everything from the RORO0100 structure plus additional information.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4), UNSIGNED	Bytes returned
4	4	BINARY(4), UNSIGNED	Bytes available
8	8	BINARY(4), UNSIGNED	Reference count
12	0C	BINARY(4), UNSIGNED	In-use indicator
16	10	BINARY(4), UNSIGNED	Offset to simple reference types
20	14	BINARY(4), UNSIGNED	Length of simple reference types
24	18	BINARY(4), UNSIGNED	Offset to extended reference types
28	1C	BINARY(4), UNSIGNED	Length of extended reference types
32	20	BINARY(4), UNSIGNED	Offset to job list
36	24	BINARY(4), UNSIGNED	Jobs returned
40	28	BINARY(4), UNSIGNED	Jobs available
Offset determined from *Offset to simple reference types* field		Qp0l_Sim_Ref_Types_Output Structure	Simple reference types structure. See Simple object reference types structure description for a description of this structure.
Offset determined from the *Offset to Extended Reference Types* field		Qp0l_Ext_Ref_Types_Output Structure	Extended reference types structure. See Extended object reference types structure description for a description of this structure. The reference counts contained within this structure represent the number of references for all jobs in the job list.
Offset determined from *Offset to Job List* field		Qp0l_Job_Using_Object Structure	Referencing job list. The Job using object structure will be repeated for each job.

Job Using Object Structure Description (*Qp0l_Job_Using_Object*)

This structure is imbedded within the RORO0200 format. It is used to return information about a job that is known to be holding a reference on the object.

Offset			

Dec	Hex	Type	Field
0	0	BINARY(4), UNSIGNED	Displacement to simple reference types
4	4	BINARY(4), UNSIGNED	Length of simple reference types
8	8	BINARY(4), UNSIGNED	Displacement to extended reference types
12	0C	BINARY(4), UNSIGNED	Length of extended reference types
16	10	BINARY(4), UNSIGNED	Displacement to next job entry
20	14	CHAR(10)	Job name
30	1E	CHAR(10)	Job user
40	28	CHAR(6)	Job number
Offset determined from the *Displacement to Simple Reference Types* field		Qp0l_Sim_Ref_Types_Output Structure	Simple reference types structure. See Simple object reference types structure description for a description of this structure.
Offset determined from the *Displacement to Extended Reference Types* field		Qp0l_Ext_Ref_Types_Output Structure	Extended reference types structure. See Extended object reference types structure description for a description of this structure. The reference counts contained within this structure represent the number of references for this specific job.

Simple Object Reference Types Structure Description (*Qp0l_Sim_Ref_Types_Output*)

This structure is imbedded within the RORO0100 and RORO0200 formats. It is used to return object reference type information.

Each binary field reference type will be set to either 0 or a positive value that represents the number of references for that type. This number will have different meanings depending on the structure it is imbedded within. When this structure is imbedded within a RORO0100 output, or imbedded within the header portion of the RORO0200 output, then these values represent the number of known references of this type. When this structure is imbedded within a specific job list entry, then these values represent the number of references for that specific type within that specific job itself.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4), UNSIGNED	Read only
4	4	BINARY(4), UNSIGNED	Write only
8	8	BINARY(4), UNSIGNED	Read/write
12	0C	BINARY(4), UNSIGNED	Execute
16	10	BINARY(4), UNSIGNED	Share with readers only
20	14	BINARY(4), UNSIGNED	Share with writers only
24	18	BINARY(4), UNSIGNED	Share with readers and writers
28	1C	BINARY(4), UNSIGNED	Share with neither readers nor writers
32	20	BINARY(4), UNSIGNED	Attribute lock
36	24	BINARY(4), UNSIGNED	Save lock

40	28	BINARY(4), UNSIGNED	Internal save lock
44	2C	BINARY(4), UNSIGNED	Link changes lock
48	30	BINARY(4), UNSIGNED	Checked out
52	34	CHAR(10)	Checked out user name
62	3E	CHAR(2)	Reserved (Binary 0)

Extended Object Reference Types Structure Description (*Qp0l_Ext_Ref_Types_Output*)

This structure is imbedded within the RORO0200 format. It is used to return object reference type information.

Each binary field reference type will be set to either 0 or a positive value that represents the number of references for that type. This number will have different meanings depending on the structure it is imbedded within. When this structure is imbedded within the header portion of the RORO0200 output, then these values represent the number of jobs in the job list that contains a reference of this type. When this structure is imbedded within a specific job list entry, then these values represent the number of references for that specific type within that specific job itself.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4), UNSIGNED	Read only, share with readers only
4	4	BINARY(4), UNSIGNED	Read only, share with writers only
8	8	BINARY(4), UNSIGNED	Read only, share with readers and writers
12	0C	BINARY(4), UNSIGNED	Read only, share with neither readers nor writers
16	10	BINARY(4), UNSIGNED	Write only, share with readers only
20	14	BINARY(4), UNSIGNED	Write only, share with writers only
24	18	BINARY(4), UNSIGNED	Write only, share with readers and writers
28	1C	BINARY(4), UNSIGNED	Write only, share with neither readers nor writers
32	20	BINARY(4), UNSIGNED	Read/write, share with readers only
36	24	BINARY(4), UNSIGNED	Read/write, share with writers only
40	28	BINARY(4), UNSIGNED	Read/write, share with readers and writers
44	2C	BINARY(4), UNSIGNED	Read/write, share with neither readers nor writers
48	30	BINARY(4), UNSIGNED	Execute, share with readers only
52	34	BINARY(4), UNSIGNED	Execute, share with writers only
56	38	BINARY(4), UNSIGNED	Execute, share with readers and writers
60	3C	BINARY(4), UNSIGNED	Execute, share with neither readers nor writers
64	40	BINARY(4), UNSIGNED	Execute/read, Share with readers only
68	44	BINARY(4), UNSIGNED	Execute/read, share with writers only
72	48	BINARY(4), UNSIGNED	Execute/read, share with readers and writers
76	4C	BINARY(4), UNSIGNED	Execute/read, share with neither readers nor writers
80	50	BINARY(4), UNSIGNED	Attribute lock
84	54	BINARY(4), UNSIGNED	Save lock

88	58	BINARY(4), UNSIGNED	Internal save lock
92	5C	BINARY(4), UNSIGNED	Link changes lock
96	60	BINARY(4), UNSIGNED	Current directory
100	64	BINARY(4), UNSIGNED	Root directory
104	68	BINARY(4), UNSIGNED	File server reference
108	6C	BINARY(4), UNSIGNED	File server working directory
112	70	BINARY(4), UNSIGNED	Checked out
116	74	CHAR(10)	Checked out user name
126	7E	CHAR(2)	Reserved (Binary 0)

Field Descriptions for RORO0100 and RORO0200 Output Structures and their Imbedded Structures

Attribute lock. Attribute changes are prevented.

Bytes available. Number of bytes of output data that was available to be returned.

Bytes returned. Number of bytes returned in the output buffer.

Checked out. Indicates whether the object is currently checked out. If it is checked out, then the *Checked Out User Name* contains the name of the user who has it checked out.

Checked out user name. Contains the name of the user who has the object checked out, when the *Checked Out* field indicates that it is currently checked out. This field is set to blanks (x'40) if the object is not checked out.

Current directory. The object is a directory that is being used as the current directory of the job.

Displacement to extended reference types. Displacement from the beginning of the structure containing this field to the beginning of the Extended Reference Types structure. If this field is 0, then no extended reference types were available to be returned, or not enough space was provided to include any portion of the Extended Reference Types structure.

Displacement to next job entry. Displacement from the beginning of the structure containing this field to the beginning of the next Job Using Object structure. If this field is 0, then there are no more jobs in the list, or not enough space was provided to include any more Job Using Object structures.

Displacement to simple reference types. Displacement from the beginning of the structure containing this field to the beginning of the Simple Reference Type structure. If this field is 0, then no simple reference types were available to be returned, or not enough space was provided to include any portion of the Simple Reference Types structure.

Execute. Execute only access.

Execute, share with readers only. Execute only access. The sharing mode allows sharing with read and execute access intents only.

Execute, share with readers and writers. Execute only access. The sharing mode allows sharing with read, execute, and write access intents.

Execute, share with writers only. Execute only access. The sharing mode allows sharing with write access

intents only.

Execute, share with neither readers nor writers. Execute only access. The sharing mode allows sharing with no other access intents.

Execute/read, share with readers only. Execute and read access. The sharing mode allows sharing with read and execute access intents only.

Execute/read, share with readers and writers. Execute and read access. The sharing mode allows sharing with read, execute, and write access intents.

Execute/read, share with writers only. Execute and read access. The sharing mode allows sharing with write access intents only.

Execute/read, share with neither readers nor writers. Execute and read access. The sharing mode allows sharing with no other access intents.

Extended reference types structure. This is a Qp0l_Ext_Ref_Types_Output structure containing fields that indicate different types of references that may be held on an object. Some of these are actually a grouping of multiple **Simple Reference Types** that were known to have been specified by the referring instance. These are not additional references; they are a redefinition of the same references described in the Simple Reference Types structure.

File server reference. The File Server is holding a generic reference on the object on behalf of a client.

File server working directory. The object is a directory, and the File Server is holding a working directory reference on it on behalf of a client.

In-use indicator The object is currently in-use. NOTE: This indicator will be set to one of the following values:

QP0L_OBJECT_NOT_IN_USE (0)

> The object is not in use and all of the reference type fields returned are 0.

QP0L_OBJECT_IN_USE (1)

> The object is in use. At least one of the reference type fields is greater than 0. This condition may occur even if the Reference Count field's value is 0.

Internal save lock. The object is being referenced internally during a save operation on a different object.

Job name. Name of the job.

Job number. Number associated with the job.

Job user. User profile associated with the job.

Jobs available. Number of referencing jobs available. This may be greater than the **Jobs Returned** field when the caller did not provide enough space to receive all of the job information.

Jobs returned. Number of referencing jobs returned in the job list.

Length of extended reference types. Length of the Extended Reference Types information.

Length of simple reference types. Length of the Simple Reference Types information.

Link changes lock. Changes to links in the directory are prevented.

Offset to extended reference types. Offset from the beginning of the *Receiver_Ptr* to the beginning of the

Extended Reference Types structure. If this field is 0, then no extended reference types were available to be returned, or not enough space was provided to include any portion of the Extended Reference Types structure.

Offset to job list. Offset from the beginning of the *Receiver_Ptr* to the beginning of the first Job Using Object structure. If this field is 0, then there are no jobs in the list.

Offset to simple reference types. Offset from the beginning of the *Receiver_Ptr* to the beginning of the Simple Reference Type structure. If this field is 0, then no simple reference types were available to be returned, or not enough space was provided to include any portion of the Simple Reference Types structure.

Read only. Read only access.

Read only, share with readers only. Read only access. The sharing mode allows sharing with read and execute access intents only.

Read only, share with readers and writers. Read only access. The sharing mode allows sharing with read, execute, and write access intents.

Read only, share with writers only. Read only access. The sharing mode allows sharing with write access intents only.

Read only, share with neither readers nor writers. Read only access. The sharing mode allows sharing with no other access intents.

Read/write. Read and write access.

Read/write, share with readers only. Read and write access. The sharing mode allows sharing with read and execute access intents only.

Read/write, share with readers and writers. Read and write access. The sharing mode allows sharing with read, execute, and write access intents.

Read/write, share with writers only. Read and write access. The sharing mode allows sharing with write access intents only.

Read/write, share with neither readers nor writers. Read and write access. The sharing mode allows sharing with no other access intents.

Reference count. Current number of references on the object. NOTE: This may be 0 even though the In-Use Indicator indicates that the object is in use.

Referencing job list. Variable length list of Qp0l_Job_Using_Object structures for jobs that are currently referencing the object.

Root directory. The object is a directory that is being used as the root directory of the job.

Save lock. The object is being referenced by an object save operation.

Share with readers only. The sharing mode allows sharing with read and execute access intents only.

Share with readers and writers. The sharing mode allows sharing with read, execute, and write access intents.

Share with writers only. The sharing mode allows sharing with write access intents only.

Share with neither readers nor writers. The sharing mode allows sharing with no other access intents.

Simple reference types structure. This is a Qp0l_Sim_Ref_Types_Output structure containing fields that indicate different types of references that may be held on an object.

Write only. Write only access.

Write only, share with readers only. Write only access. The sharing mode allows sharing with read and execute access intents only.

Write only, share with readers and writers. Write only access. The sharing mode allows sharing with read, execute, and write access intents.

Write only, share with writers only. Write only access. The sharing mode allows sharing with write access intents only.

Write only, share with neither readers nor writers. Write only access. The sharing mode allows sharing with no other access intents.

Error Messages

Message ID	Error Message Text
CPF3C21 E	Format name &1 is not valid.
CPF3C24 E	Length of the receiver variable is not valid.
CPF3C36 E	Number of parameters, &1, entered for this API was not valid.
CPF3CF1 E	Error code parameter not valid.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. Since both available formats are variable length, following are the recommended minimum lengths pertaining to their corresponding formats:

    - RORO0100: The size of a RORO0100 Output structure plus the size of a *Simple Reference Types* structure.

    - RORO0200: This structure varies dynamically, and therefore there is no formula that can yield a size large enough to always retrieve all of the available information. However, programs may consider first calling QP0LROR with the RORO0100 format. This will quickly return the number of references currently on the object. Then the program could allocate a buffer equal in size to: size of a *Job Using Object* structure (including the size of both the Simple and Extended Reference Type structures) multiplied by the number of references, and then add the sizes of a RORO0100 output, RORO0200 output, and Simple Reference Types structures. Now the program could call QP0LROR with the RORO0200 format requested and the computed size.

      If the RORO0200 format was specified, but there was not enough space provided to

receive a complete list of job information, then only those job entries that completely fit in the buffer will be returned. The RORO0200 output structure contains a field called *JobsAvailable* that will always contain the total number of referencing jobs that were available for returning to the caller at that instance in time.

Notes

- There are no locks obtained on the object while this API is running. Therefore, when this API is used on an object that is actively in use (for example, its lock and reference state is changing while this API is running), some fields in the returned information may be inconsistent with other fields returned on the same invocation of QP0LROR.
- The number of references on the object may change between multiple calls to this API. Therefore, the above formula for calculating output buffer size for a RORO0200 format may not be enough space under all conditions.
- There are some reference types that are obtained on the object without incrementing the object's reference count. This could result in a reference count of zero while the object contains reference types. In this instance, the above formula for calculating output buffer size for a RORO0200 format may not be enough space.

2. The list of simple object reference types in the base portions of the RORO0100 and RORO0200 output structures may not contain complete information for objects residing in file systems other than the Root ('/'), QOpenSys, and User-defined file systems. The simple reference types will, however, be set in the job array elements in the RORO0200 output structure for any file system.

3. The list of object reference types in the RORO0200 output formats may be an incomplete list of references for objects residing in file systems other than the Root ('/'), QOpenSys, and User-defined file systems. Objects in some of the other file systems can be locked with interfaces that do not use the Integrated File System. Therefore, references returned by this API will only be references that were obtained as part of an Integrated File System operation, or an operation that cause the Integrated File System operation to occur.

4. Under some circumstances, the list of jobs that are referencing the object may be incomplete. However, jobs not listed in the job list may still have their references listed in the RORO0100 output. This occurs when system programs obtain references directly on an object without obtaining an open descriptor for the object.

5. At some instances during the save or restore of an Integrated File System object, the object may have references held by the job even though its reference count is 0.

6. File systems that access remote objects, such as Network File System (NFS) and the QFileSvr.400 file systems, will only be returning references that are locally obtained on the object. Any references that the remote system may have on the remote object are not returned by this API.

7. This type of reference information is also viewable through the iSeries Navigator application. The terminology, however, differs in that iSeries Navigator refers to this type of information as "Usage" information instead of "Reference" information.

Related Information

- The <**qp0lror.h**> file (see Header Files for UNIX-Type Functions)

Example

The following is an example use of this API.

See Code disclaimer information for information pertaining to code examples.

```
#include <qp0lror.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void main()
{
    struct PathNameStruct
    {
 Qlg_Path_Name_T header;
 char p[50];
    };

    struct PathNameStruct path;

    char pathName[] = "/CustomerData";

    Qus_EC_t errorCode;

    /* Define a constant for the number of output buffer bytes
       provided for the RORO0100 format.                       */
#define OUTPUT_BYTES_RORO0100                                   \
             (sizeof(Qp0l_RORO0100_Output_T) +                  \
              sizeof(Qp0l_Sim_Ref_Types_Output_T) +    \
              100)       /* Pad space for potential gap between
                            the 2 structures.                  */

    /* Declare some space for the RORO0100 output.             */
    char output100Buf[OUTPUT_BYTES_RORO0100];

    /* Declare a pointer for retrieving the RORO0100 format.   */
    Qp0l_RORO0100_Output_T *output100P;

    /* Declare a pointer to retrieve the RORO0200 format.      */
    Qp0l_RORO0200_Output_T *output200P;

    /* Declare a job using object pointer.                     */
    Qp0l_Job_Using_Object_T *jobP;

    unsigned outputBufSize;

    /* Set output buffer pointer and length for retrieving the
       RORO0100 format.                                        */
```

```
   output100P = (Qp0l_RORO0100_Output_T *)output100Buf;

   /* Setup the object's path name structure.                        */
   memset(&path, 0, sizeof(path));
   path.header.CCSID = 37;
   memcpy(path.header.Country_ID,"US",2);
   memcpy(path.header.Language_ID,"ENU",3);
   path.header.Path_Type = QLG_CHAR_SINGLE;
   path.header.Path_Length = strlen(pathName);
   path.header.Path_Name_Delimiter[0] = '/';
   memcpy(path.p, pathName, path.header.Path_Length);

   /* Setup the error code structure to cause the error to be
      returned within the error structure.                           */
   errorCode.Bytes_Provided = sizeof(errorCode);
   errorCode.Bytes_Available = 0;

   /* First call QP0LROR to get the short format.  We will
      use that information about references to conditionally
      allocate more space and then get the longer
      running format's information.                                  */
   QP0LROR(output100P,
    OUTPUT_BYTES_RORO0100,
    QP0LROR_RORO0100_FORMAT,
    (Qlg_Path_Name_T *) &path,
    &errorCode);

   /* Check if an error occurred.                                    */
   if (errorCode.Bytes_Available != 0)
   {
printf("Error occurred for RORO0100.\n");
return;
   }

   /* Check if we received any references that might be
      associated with a job.  If not, return.                        */
   if (output100P->Count == 0)
   {
printf("QP0LROR returned a reference count of %d\n",
       output100P->Count);
return;
   }

   /* If we get here, then we have at least 1 reference that
      may be identifiable to a job.  We will call the
      QP0LROR API to get the RORO0200 format.  First we
      compute a buffer size to use.  Note:  this calculation
      sums up the sizes of all structures contained within
      the RORO0200 format, but doesn't consider gaps between
      each of the structure.  To attempt to cover potential
      gaps between structures, an extra 1000 bytes is being
      allocated and room for 10 additional jobs.                     */
   outputBufSize =
     sizeof(Qp0l_RORO0200_Output_T) +
     sizeof(Qp0l_Sim_Ref_Types_Output_T) +
     sizeof(Qp0l_Ext_Ref_Types_Output_T) +
     ((output100P->Count + 10) *
```

```
      (sizeof(Qp0l_Job_Using_Object_T) +
sizeof(Qp0l_Sim_Ref_Types_Output_T) +
sizeof(Qp0l_Ext_Ref_Types_Output_T)
) + 1000
      );

   if (NULL == (output200P =
  (Qp0l_RORO0200_Output_T *)malloc(outputBufSize)))
   {
printf("No space available.\n");
return;
   }

   /* Retrieve object references.                           */
   QP0LROR(output200P,
    outputBufSize,
    QP0LROR_RORO0200_FORMAT,
    (Qlg_Path_Name_T *) &path,
    &errorCode);

   /* Check if an error occurred.                           */
   if (errorCode.Bytes_Available != 0)
   {
free(output200P);
printf("Error occurred for RORO0200.\n");
return;
   }

   /* If there was more information available than we had
      provided receiver space for, then we will allocate a
      larger buffer and try once again.  This could potentially
      keep reoccurring, but this example will stop after this
      second retry.                                         */
   if (output200P->BytesReturned < output200P->BytesAvailable)
   {
       /* Use the bytes available value to determine how much
          more buffer size is needed.  We will pad it with an
          extra 1000 bytes to try and handle more jobs obtaining
          references between calls to QP0LROR.              */
       outputBufSize = output200P->BytesAvailable + 1000;

if (NULL == (output200P = (Qp0l_RORO0200_Output_T *)
      realloc((void *)output200P,
       outputBufSize)))
{
    printf("No space available.\n");
    return;
}

QP0LROR(output200P,
 outputBufSize,
 QP0LROR_RORO0200_FORMAT,
 (Qlg_Path_Name_T *) &path,
 &errorCode);

       /* Check if an error occurred.                       */
if (errorCode.Bytes_Available != 0)
```

```
 {
     free(output200P);
     printf("Error occurred for RORO0200 (2nd call).\n");
     return;
 }
    }

    /* Print some output.                                         */
    printf("Reference count:  %d\n",output200P->Count);
    printf("Jobs returned:    %d\n",output200P->JobsReturned);

    if (output200P->JobsReturned > 0)
    {
 jobP = (Qp0l_Job_Using_Object_T *)
   ((char *)output200P + output200P->JobsOffset);
 printf("First job's name: %10.10s %10.10s %6.6s",
        jobP->Name,
        jobP->User,
        jobP->Number);
    }

    free(output200P);

    return;
}
```

Example Output:

```
Reference count:  1
Jobs returned:    1
First job's name: JOBNAME123 JOBUSER123 123456
```

«

API introduced: V5R2

Qp0lSaveStgFree()--Save Storage Free

Syntax

```
#include <Qp0lstdi.h>

int Qp0lSaveStgFree(
  Qlg_Path_Name_T            *Path_Name,
  Qp0l_StgFree_Function_t    *UserFunction_ptr,
  void                       *Function_CtlBlk_ptr);
```

Service Program Name: QP0LLIB3

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **Qp0lSaveStgFree()** function calls a user-supplied exit program to save OS/400 objects of type *STMF and, upon successful completion of the exit program, frees the storage for the object and marks the object as storage freed. The *STMF object and its attributes remain on the system, but the storage occupied by the *STMF object's data is deleted. The *STMF object cannot be used until it is restored to the system. This is accomplished by either of the following:

- Restoring the object using the RST command.
- Requesting an operation on the object, requiring one of the following, which will dynamically retrieve (restore) the *STMF object:
  - Accessing the object's data (**open()**, **creat()**, MOV, CPY, CPYFRMSTMF, or CPYTOSTMF).
  - Adding a new name to the object (RNM, ADDLNK, **link()**, **rename()**, **Qp0lRenameKeep()**, or **Qp0lRenameUnlink()**).
  - Checking out the object (CHKOUT).

The restore operation is done by calling a user-provided exit program registered against the Storage Extension exit point QIBM_QTA_STOR_EX400. For information on this exit point, see the Storage Extension Exit Program.

Qp0lSaveStgFree() returns EOFFLINE for an object that is already storage freed or returns EBUSY for an object that is checked out.

The user exit program can be either a procedure or a program.

Parameters

Path_Name

(Input) A pointer to a path name whose last component is the object that is saved and whose storage is freed. This path name is in the Qlg_Path_Name_T format. For more information on this structure, see Path name format.

If the last component of the path name supplied on the call to **Qp0lSaveStgFree()** is a symbolic link, then **Qp0lSaveStgFree()** resolves and follows the link to its target and performs its normal **Qp0lSaveStgFree()** functions on that target. If the symbolic link refers to an object in a remote file system, **Qp0lSaveStgFree()** returns ENOTSUP to the calling program.

UserFunction_ptr

(Input) A pointer to a structure that contains information about the user exit program that the caller wants **Qp0lSaveStgFree()** to call to save an *STMF object. This user exit program can be either a procedure or a program. If this pointer is NULL, **Qp0lSaveStgFree()** does not call an exit program to save the object but does free the object's storage and marks it as storage freed.

User Function Pointer			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Function type flag
14	E	CHAR(10)	Program library
4	4	CHAR(10)	Program name
24	18	CHAR(1)	Multithreaded job action
25	19	CHAR(7)	Reserved
32	20	PP(*)	Procedure pointer to exit procedure

Function type flag. A flag that indicates whether the Save Storage Free exit program called by **Qp0lSaveStgFree()** is a procedure or a program. If the exit program is a procedure, this flag is set to 0, and the procedure pointer to exit procedure field points to the procedure called by **Qp0lSaveStgFree()**. If the exit program is a program, this flag is set to 1 and a program name and program library are provided, respectively, in the program name and program library fields. Valid values follow:

0 QP0L_USER_FUNCTION_PTR: A user procedure is called.

1 QP0L_USER_FUNCTION_PGM: A user program is called.

Multithreaded job action. (Input) A CHAR(1) value that indicates the action to take in a multithreaded job. The default value is QP0L_MLTTHDACN_SYSVAL. For release compatibility and for processing this parameter against the QMLTTHDACN system value, x'00, x'01', x'02', & x'03' are treated as x'F0', x'F1', x'F2', and x'F3'.

x'00' QP0L_MLTTHDACN_SYSVAL: The API evaluates the QMLTTHDACN system value to determine the action to take in a multithreaded job. Valid QMLTTHDACN system values follow:

'1' Call the exit program. Do not send an informational message.

'2' Call the exit program and send informational message CPI3C80.

'3' The exit program is not called when the API determines that it is running in a multithreaded job. ENOTSAFE is returned.

x'01' QP0L_MLTTHDACN_NOMSG: Call the exit program. Do not send an informational message.

x'02' QP0L_MLTTHDACN_MSG: Call the exit program and send informational message CPI3C80.

x'03' QP0L_MLTTHDACN_NO: The exit program is not called when the API determines that it is running in a multithreaded job. ENOTSAFE is returned.

Procedure pointer to exit procedure. If the function type flag is 0, which indicates that a procedure is called instead of a program, this field contains a procedure pointer to the procedure that **Qp0lSaveStgFree()** calls. This field must be NULL if the function type flag is 1.

Program library. If the function type flag is 1, indicating a program is called, this field contains the library in which the program being called (identified by the program name field) is located. This field must be blank if the function type flag is 0.

Program name. If the function type flag is 1, indicating a program is called, this field contains the name of the program that is called. The program should be located in the library identified by the program library field. This field must be blank if the function type flag is 0.

Reserved. A reserved field. This field must be set to binary zero.

Function_CtlBlk_ptr

(Input) A pointer to any data that the caller of **Qp0lSaveStgFree()** wants to have passed to the user-defined Save Storage Free exit program that **Qp0lSaveStgFree()** calls to save an *STMF object. **Qp0lSaveStgFree()** does not process the data that is referred to by this pointer. The API passes this pointer as a parameter to the user-defined Save Storage Free exit program that was specified on its call. This is a means for the caller of **Qp0lSaveStgFree()** to pass information to and from the Save Storage Free exit program.

Authorities

The following table shows the authorization required for the Qp0lSaveStgFree() API.

Object Referred to	Authority Required	errno
Each directory, preceding the last component, in a *path name*	*RX	EACCES
Object	*SAVSYS or *RW	EACCES
Any called program pointed to by the *UserFunction_ptr* parameter	*X	EACCES
Any library containing the called program pointed to by the *UserFunction_ptr* parameter	*X	EACCES

Return Value

0 **Qp0lSaveStgFree()** was successful.

-1 **Qp0lSaveStgFree()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **Qp0lSaveStgFree()** is not successful, *errno* indicates one of the following errors:

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[ENOSYSRSC]

System resources not available to complete request.

[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.

[EOFFLINE]

Object 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 successfully restored. The object's data was saved and freed either by saving the object with the STG(*FREE) parameter, or by calling an API.

[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
CPI3C80 I	An exit program has been called for which the threadsafety status was not known.
CPFA0D4 E	File system error occurred.
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.

Usage Notes

- 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
    - QOPT
- If the Save Storage Free exit program calls the SAV command or the **QsrSave** function or any other function that is not threadsafe, and there are secondary threads active in the job, **Qp0lSaveStgFree()** may fail as a result.
- If the Save Storage Free exit program is not threadsafe or uses a function that is not threadsafe, then **Qp0lSaveStgFree()** is not threadsafe.

Related Information

- The <**Qp0lstdi.h**> file
- QlgSaveStgFree()--Save Storage Free (using NLS-enabled path name)
- Save Storage Free Exit Program

Example

See **Qp0lGetAttr()** description for a code example that shows a call to **Qp0lSaveStgFree()** by using a procedure as the exit program. This API also shows an example of a call to **Qp0lGetAttr()**.

API introduced: V4R3

Qp0lSetAttr()--Set Attributes

Syntax

```
#include <Qp0lstdi.h>
int Qp0lSetAttr
       (Qlg_Path_Name_T      *Path_Name,
        char                 *Buffer_ptr,
        uint                  Buffer_Size,
        uint                  Follow_Symlnk, ...);
```

Service Program Name: QP0LLIB3

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **Qp0lSetAttr**() function sets one of a set of defined attributes, on each call, for the object that is referred to by the input **Path_Name*. The object must exist, the user must have authority to it, and the attribute must be supported by the file system to which the object belongs. When an attribute is not supported by the file system, **Qp0lSetAttr()** will fail with ENOTSUP. See the Usage Notes section for more information.

If the last component of the *Path_Name* parameter is a symbolic link, the **Qp0lSetAttr()** either sets the attribute of the symbolic link or sets the attribute of the object that the symbolic link names. This depends on the value of the *Follow_Symlnk* parameter.

All times that are set by **Qp0lSetAttr**() are in seconds since the Epoch so that they are consistent with UNIX-type APIs. The Epoch is the time 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time. If the OS/400 date is set prior to 1970, all time values will be zero.

Parameters

Path_Name

(Input) The path name of the object for which attribute information is set. This path name is in the Qlg_Path_Name_T format. For more information on this structure, see Path name format.

Buffer_ptr

(Input) A pointer to a buffer containing a constant that identifies the attribute and the value for the attribute that **Qp0lSetAttr()** sets. The number of bytes allocated for this buffer is in the *Buffer_Size* parameter.

The following table describes the format of the entry in the buffer.

Buffer Pointer			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Offset to next attribute entry

4	4	BINARY(4)	Attribute identification
8	8	BINARY(4)	Size of attribute data
12	C	CHAR(4)	Reserved
16	10	CHAR(*)	Attribute data

Attribute data. The value to which the attribute is set.

Attribute identification. The constant identifying the attribute being set. Valid values are:

4 QP0L_ATTR_CREATE_TIME: (UNSIGNED (BINARY(4)) The time the object was created.

5 QP0L_ATTR_ACCESS_TIME: (UNSIGNED (BINARY(4)) The time the object's data was last accessed.

7 QP0L_ATTR_MODIFY_TIME: (UNSIGNED (BINARY(4)) The time the object's data was last changed.

17 QP0L_ATTR_PC_READ_ONLY: (CHAR(1)) Whether the object can be written to or deleted, have its extended attributes changed or deleted, or have its size changed. Valid values are:

x'00' QP0L_PC_NOT_READONLY: The object can be changed.

x'01' QP0L_PC_READONLY: The object cannot be changed.

18 QP0L_ATTR_PC_HIDDEN: (CHAR(1)) Whether the object can be displayed using an ordinary directory listing.

x'00' QP0L_PC_NOT_HIDDEN: The object is not hidden.

x'01' QP0L_PC_HIDDEN: The object is hidden.

19 QP0L_ATTR_PC_SYSTEM: (CHAR(1)) Whether the object is a system file and is excluded from normal directory searches.

x'00' QP0L_PC_NOT_SYSTEM: The object is not a system file.

x'01' QP0L_PC_SYSTEM: The object is a system file.

20 QP0L_ATTR_PC_ARCHIVE: (CHAR(1)) Whether the object has changed since the last time the file was saved or reset by a PC client.

x'00' QP0L_PC_NOT_CHANGED: The object has not changed.

x'01' QP0L_PC_CHANGED: The object has changed.

21 QP0L_ATTR_SYSTEM_ARCHIVE: (CHAR(1)) Whether the object has changed and needs to be saved. It is set on when an object's change time is updated, and set off when the object has been saved.

- *x'00'* QP0L_SYSTEM_NOT_CHANGED: The object has not changed and does not need to be saved.
- *x'01'* QP0L_SYSTEM_CHANGED: The object has changed and does need to be saved.

22 QP0L_ATTR_CODEPAGE: (BINARY(4)) The code page used to derive a coded character set identifier (CCSID) used for the data in the file or the extended attributes of the directory.

26 QP0L_ATTR_ALWCKPWRT: (CHAR(1)) Whether a stream file (*STMF) can be shared with readers and writers during the save-while-active checkpoint processing. Setting this attribute may cause unexpected results. Please refer to the Backup and Recovery book for details on this attribute.

- *x'00'* QP0L_NOT_ALWCKPWRT: The object can be shared with readers only.
- *x'01'* QP0L_ALWCKPWRT: The object can be shared with readers and writers.

27 QP0L_ATTR_CCSID: (BINARY(4)) The CCSID of the data and extended attributes of the object.

31 QP0L_ATTR_DISK_STG_OPT (CHAR(1)) Which option should be used to determine how auxiliary storage is allocated by the system for the specified object. The option will take effect immediately and be part of the next auxiliary storage allocation for the object. This option can only be specified for byte stream files in the Root (/), QOpensys and User-defined file systems. This option will be ignored for *TYPE1 byte stream files. Valid values are:

- *x'00'* QP0L_STG_NORMAL: The auxiliary storage will be allocated normally. That is, as additional auxiliary storage is required, it will be allocated in logically sized extents to accomodate the current space requirement, and anticipated future requirements, while minimizing the number of disk I/O operations. If the QP0L_ATTR_DISK_STG_OPT attribute has not been specified for an object, this value is the default.
- *x'01'* QP0L_STG_MINIMIZE: The auxiliary storage will be allocated to minimize the space used by the object. That is, as additional auxiliary storage is required, it will be allocated in small sized extents to accomodate the current space requirement. Accessing an object composed of many small extents may increase the number of disk I/O operations for that object.
- *x'02'* QP0L_STG_DYNAMIC: The system will dynamically determine the optimum auxiliary storage allocation for the object, balancing space used versus disk I/O operations. For example, if a file has many small extents, yet is frequently being read and written, then future auxiliary storage allocations will be larger extents to minimize the number of disk I/O operations. Or, if a file is frequently truncated, then future auxiliary storage allocations will be small extents to minimize the space used. Additionally, information will be maintained on the byte stream file sizes for this system and its activity. This file size information will also be used to help determine the optimum auxiliary storage allocations for this object as it relates to the other objects sizes.

32 QP0L_ATTR_MAIN_STG_OPT: (CHAR(1)) Which option should be used to determine how main storage is allocated and used by the system for the specified object. The option will take effect the next time the specified object is opened. This option can only be specified for byte stream files in the Root (/), QOpensys and User-defined file systems. Valid values are:

x'00' QP0L_STG_NORMAL: The main storage will be allocated normally. That is, as much main storage as possible will be allocated and used. This minimizes the number of disk I/O operations since the information is cached in main storage. If the QP0L_ATTR_MAIN_STG_OPT attribute has not been specified for an object, this value is the default.

x'01' QP0L_STG_MINIMIZE: The main storage will be allocated to minimize the space used by the object. That is, as little main storage as possible will be allocated and used. This minimizes main storage usage while increasing the number of disk I/O operations since less information is cached in main storage.

x'02' QP0L_STG_DYNAMIC: The system will dynamically determine the optimum main storage allocation for the object depending on other system activity and main storage contention. That is, when there is little main storage contention, as much storage as possible will be allocated and used to minimize the number of disk I/O operations. And when there is significant main storage contention, less main storage will be allocated and used to minimize the main storage contention.«

200 QP0L_ATTR_RESET_DATE: (UNSIGNED (BINARY(2)) The count of the number of days an object has been used. Usage has different meanings according to the file system and according to the individual object types supported within a file system. Usage can indicate the opening or closing of a file or can refer to adding links, renaming, restoring, or checking out an object. The usage information format is defined in the Qp0lstdi.h header file as data type Qp0l_Usage_t and is shown in the following table. This attribute can be set to zero only. An attempt to set to any other value will result in *errno* [EINVAL].

When this attribute is set, the date use count reset for the object is set to the current date.

»*300* QP0L_ATTR_SUID: (CHAR(1)) Set effective user ID (UID) at execution time. This value is ignored if the specified object is a directory. Valid values are:

x'00' QP0L_SUID_OFF: The user ID (UID) is not set at execution time.

x'01' QP0L_SUID_ON: The object owner is the effective user ID (UID) at execution time.

301 QP0L_ATTR_SGID: (CHAR(1)) Set effective group ID (GID) at execution time. Valid values are:

x'00' QP0L_SGID_OFF: If the object is a file, the group ID (GID) is not set at execution time. If the object is a directory in the Root ('/'), QOpensys, and user-defined file systems, the group ID (GID) of objects created in the directory is set to the effective GID of the thread creating the object. This value cannot be set for other file systems.

x'01' QP0L_SGID_ON: If the object is a file, the group ID (GID) is set at execution time. If the object is a directory, the group ID (GID) of objects created in the directory is set to the GID of the parent directory.

Offset to next attribute entry. (Output) This field is not used by the **Qp0lSetAttr()** function. It is provided for alignment so that the same buffer format returned from the **Qp0lGetAttr()** function can be used as input to the **Qp0lSetAttr()** function.

Reserved. A reserved field. This field must be set to binary zero.

Size of attribute data. The exact size of the data for this attribute. If this size does not match the size that the system stores for this attribute, [EINVAL] is returned.

Buffer_Size

(Input) The size in bytes of the buffer pointed to by the *Buffer_ptr* parameter.

Follow_Symlnk

(Input) If the last component in the **Path_Name* is a symbolic link, **Qp0lSetAttr()** either acts upon the symbolic link or the path contained in the symbolic link. This depends on the value of the *Follow_Symlnk* parameter. Valid values are:

0 QP0L_DONOT_FOLLOW_SYMLNK: A symbolic link in the last component is not followed. Attributes of the symbolic link object are set.

1 QP0L_FOLLOW_SYMLNK: A symbolic link in the last component is followed. The attributes of the object contained in the symbolic link are set.

Authorities

Note: Adopted authority is not used.

Authorization Required for Qp0lSetAttr() (excluding QSYS.LIB and independent ASP QSYS.LIB)

Object Referred to	Authority Required	errno
Each directory, preceding the last component, in the *path name*	*X	EACCES
Object, when setting the QP0L_ATTR_DAYS_USED_COUNT, QP0L_ATTR_ALWCKPWRT, QP0L_ATTR_DIST_STG_OPT or QP0L_ATTR_MAIN_STG_OPT attribute	*OBJMGT	EACCES
Object, when setting the QP0L_ATTR_CREATE_TIME, QP0L_ATTR_ACCESS_TIME, or QP0L_ATTR_MODIFY_TIME attribute to the current time	Owner or *W (See **Note**)	EACCES

Object, when setting the QP0L_ATTR_SUID or QP0l_ATTR_SGID values	Owner (See **Note**)	EACCES
Object, when setting the QP0L_ATTR_CREATE_TIME, QP0L_ATTR_ACCESS_TIME, or QP0L_ATTR_MODIFY_TIME attribute to a specific time	*W	EPERM
Object, when setting any other attribute	*W	EACCES
Note: If the file system supports *ALLOBJ special authority and if you have *ALLOBJ special authority, you do not need the listed object authority.		

Authorization Required for Qp0lSetAttr() (QSYS.LIB and independent ASP QSYS.LIB)		
Object Referred to	**Authority Required**	**errno**
Each directory, preceding the last component, in the *path name*	*X	EACCES
Object, when setting the QP0L_ATTR_DAYS_USED_COUNT attribute and the object type is *FILE	*OBJOPR and *OBJMGT	EACCES
Object, when setting the QP0L_ATTR_DAYS_USED_COUNT attribute and the object is a database file member	*X and *OBJMGT	EACCES
Object, when setting the QP0L_ATTR_DAYS_USED_COUNT attribute and the object is neither a *FILE object type nor a database file member	*OBJMGT	EACCES

Return Value

0 The **Qp0lSetAttr() API** was successful.

-1 The **Qp0lSetAttr() API** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If the **Qp0lSetAttr() API** is not successful, *errno* indicates one of the following errors:

[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.

[ECANCEL]

Operation canceled.

[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.

[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.

[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.

[EOFFLINE]

Object 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 successfully restored. The object's data was saved and freed either by saving the object with the STG(*FREE) parameter, or by calling an API.

[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.

[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
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.
CPE3418 E	Possible APAR condition or hardware failure.

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 QP0L_ATTR_CREATE_TIME and QP0L_ATTR_DAYS_USED_COUNT attributes are supported for objects of type *STMF only. Attempts to set them on other objects will result in the operation failing with *errno* set to [ENOTSUP].
3. QSYS.LIB »and Independent ASP QSYS.LIB «File System Differences

   The following attribute may be set on objects in these file system:
   - QP0L_ATTR_DAYS_USED_COUNT

   Attempting to set any other attribute will result in the operation failing with *errno* set to

[ENOTSUP].

When you set the QP0L_ATTR_DAYS_USED_COUNT attribute of a database file, all members in that file will have their days used count reset to 0 also.

4. Network File System Differences

   When you set the following attributes on objects in the Network File System, the operation will fail with the *errno* set to [ENOTSUP] if the attribute is not set to the following attribute value.

   - If set, QP0L_ATTR_PC_READ_ONLY must be set to an attribute value of QP0L_PC_NOT_READ_ONLY.
   - If set, QP0L_ATTR_PC_HIDDEN must be set to an attribute value of QP0L_PC_NOT_HIDDEN.
   - If set, QP0L_ATTR_PC_SYSTEM must be set to an attribute value of QP0L_PC_NOT_SYSTEM.
   - If set, QP0L_ATTR_PC_ARCHIVE must be set to an attribute value of QP0L_PC_NOT_CHANGED; however, if the object is of type *STMF, the attribute value must be QP0L_PC_CHANGED.
   - If set, QP0L_ATTR_SYSTEM_ARCHIVE must be set to an attribute value of QP0L_SYSTEM_NOT_CHANGED.

   The QP0L_ATTR_CREATE_TIME, QP0L_ATTR_DAYS_USED_COUNT, QP0L_ATTR_CODEPAGE, and QP0L_ATTR_CCSID attributes cannot be set on objects within the Network File System or they will result in the operation failing with *errno* set to [ENOTSUP].

5. QNetWare File System Differences

   The QNetWare File System does not support setting QP0L_ATTR_SYSTEM_ARCHIVE or QP0L_ATTR_DAYS_USED_COUNT. If you set any attribute on a NetWare Directory Services (NDS) object, the operation will fail with *errno* set to [ENOTSUP].

Related Information

- The <**Qp0lstdi.h**> file (see Header Files for UNIX-Type Functions)
- The <**qlg.h**> file (see Header Files for UNIX-Type Functions)
- chmod()--Change File Authorizations
- QlgSetAttr()--Set Attributes (using NLS-enabled path name)
- Qp0lGetAttr()--Get Attributes

Example

The following is an example showing a call to the **Qp0lSetAttr()** and the **Qp0lGetAttr()** APIs.

See Code disclaimer information for information pertaining to code examples.

```
 /**************************************************************/
#include "Qp0lstdi.h"
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/types.h>


 int GetAttrObject(
                  Qlg_Path_Name_T *Pathname_ptr,
                  char            *Buffer_ptr,
                  unsigned int    Buffer_size)
 {
  /**************************************************************/
  /*   Local variables                                          */
  /**************************************************************/
   struct attrStruct
   {
     Qp0l_AttrTypes_List_t  attr_struct;
     uint                   AttrTypes[10];
   };
   struct attrStruct  Attr_types_ptr;

   unsigned int buff_size_needed;
   unsigned int num_bytes_returned;
   unsigned int follow_sym;
   int          rc;

  /**************************************************************/
  /* Start of executable code                                   */
  /**************************************************************/

  /**************************************************************/
  /* Initialize Get Attributes Parameters                       */
  /**************************************************************/
   memset((void *)&Attr_types_ptr, 0x00,sizeof(struct attrStruct));
   Attr_types_ptr.attr_struct.Number_Of_ReqAttrs = 3;
   Attr_types_ptr.AttrTypes[0] = QP0L_ATTR_PC_READ_ONLY;
   Attr_types_ptr.AttrTypes[1] = QP0L_ATTR_PC_HIDDEN;
   Attr_types_ptr.AttrTypes[2] = QP0L_ATTR_CODEPAGE;
   buff_size_needed = 0;
   follow_sym = QP0L_FOLLOW_SYMLNK;

  /**************************************************************/
  /* Call Qp0lGetAttr() to retrieve attributes.                 */
  /**************************************************************/
   rc = Qp0lGetAttr(Pathname_ptr,
                    (Qp0l_AttrTypes_List_t *)&Attr_types_ptr,
                    Buffer_ptr,
                    Buffer_size,
```

```
                    &buff_size_needed,
                    &num_bytes_returned,
                    follow_sym);

  if((rc == 0) &&               /* If successful, but           */
     (num_bytes_returned <= 0)) /* Incorrect bytes returned     */
    rc = EUNKNOWN;              /* Unknown error                */

  return(rc);
}                               /* End GetAttrObject()          */


int SetAttrObject(
                  Qlg_Path_Name_T *Pathname_ptr,
                  char            *Buffer_ptr,
                  unsigned int    Buffer_size)
{

 /*****************************************************************/
 /*   Local variables                                             */
 /*****************************************************************/

  unsigned int        follow_sym;
  int                 rc;
  int                 done = 0;
  unsigned int        attrSize;
  Qp0l_Attr_Header_t *attrPtr;

 /*****************************************************************/
 /* Start of executable code                                      */
 /*****************************************************************/

 /*****************************************************************/
 /* Initialize Set Attributes Parameters                          */
 /*****************************************************************/
  follow_sym = QP0L_FOLLOW_SYMLNK;

 /*****************************************************************/
 /* Qp0lSetAttr() only sets one attribute at a time.  The         */
 /* buffer from Qp0lGetAttr may contain more than one             */
 /* attribute to set.  We may have to call Qp0lSetAttr()          */
 /* multiple times.  The Next_Attr_Offset value is the key.       */
 /* If it is greater than zero, then there is another             */
 /* attribute in the buffer.  Also, it is important to note       */
 /* that the value stored there is the offset from the start      */
 /* of the buffer, not the offset from the start of the           */
 /* current entry.                                                */
 /*****************************************************************/
  attrPtr = (Qp0l_Attr_Header_t *)Buffer_ptr;
  while(!done)
  {
      attrSize = attrPtr->Attr_Size +
        sizeof(Qp0l_Attr_Header_t);  /* Calculate attr size    */
      /***********************************************************/
      /* Call Qp0lSetAttr() to set the attribute                 */
      /***********************************************************/
      rc=Qp0lSetAttr(Pathname_ptr,
```

```
                     (char *)attrPtr,
                     attrSize,
                     follow_sym);
      if(rc != 0)              /* If the function failed       */
        done = 1;              /* End the loop                 */
      else if(attrPtr->Next_Attr_Offset > 0) /* If more data   */
        attrPtr = (Qp0l_Attr_Header_t *) /* Set attribute      */
          (Buffer_ptr + attrPtr->Next_Attr_Offset); /* pointer */
      else                     /* No more data                 */
        done = 1;              /* End the loop                 */
  }
  return(rc);
}                              /* End SetAttrObject()          */


int main (int argc, char *argv[])
{
  #define MYPN "FRED"
  #define MYPN2 "FRED2"
 /*****************************************************************/
 /*   Local variables                                             */
 /*****************************************************************/
  const char US_const[3] = "US";
  const char Language_const[4] = "ENU";
  const char Path_Name_Del_const[2] = "/";

  typedef struct pnstruct
  {
   Qlg_Path_Name_T  qlg_struct;
   char             pn[sizeof(MYPN)];
  } ;

  typedef struct pnstruct2
  {
   Qlg_Path_Name_T  qlg_struct;
   char             pn[sizeof(MYPN2)];
  } ;

  struct pnstruct pns;
  struct pnstruct2 pns2;
  int rc;

  char  BufferArea[250];
  unsigned int buffer_size = 250;

 /*****************************************************************/
 /* Start of executable code                                      */
 /*****************************************************************/

 /*****************************************************************/
 /* Initialize Pathname for original object                       */
 /*****************************************************************/
  memset((void *)&pns, 0, sizeof(struct pnstruct));
  pns.qlg_struct.CCSID = 37;
  memcpy(pns.qlg_struct.Country_ID,US_const,2);
  memcpy(pns.qlg_struct.Language_ID,Language_const,3);;
  pns.qlg_struct.Path_Type = 0;
```

```
  pns.qlg_struct.Path_Length = sizeof(MYPN) - 1;
  memcpy(pns.qlg_struct.Path_Name_Delimiter,Path_Name_Del_const,1);
  memcpy(pns.pn,MYPN,sizeof(MYPN));

 /***************************************************************/
 /* Call GetAttrObject to retrieve attributes from the source   */
 /* object.                                                     */
 /***************************************************************/
  rc = GetAttrObject((Qlg_Path_Name_T *)&pns,
                     BufferArea,
                     buffer_size);
  if (rc == 0)                  /* If GetAttr succeeded         */
  {
   /*************************************************************/
   /* Initialize Pathname for target object                     */
   /*************************************************************/
    memset((void *)&pns2, 0, sizeof(struct pnstruct2));
    pns2.qlg_struct.CCSID = 37;
    memcpy(pns2.qlg_struct.Country_ID,US_const,2);
    memcpy(pns2.qlg_struct.Language_ID,Language_const,3);;
    pns2.qlg_struct.Path_Type = 0;
    pns2.qlg_struct.Path_Length = sizeof(MYPN2)-1;
    memcpy(pns2.qlg_struct.Path_Name_Delimiter,Path_Name_Del_const,1);
    memcpy(pns2.pn,MYPN2,sizeof(MYPN2));

   /*************************************************************/
   /* Call SetAttrObject to set attributes on the target        */
   /* object.                                                   */
   /*************************************************************/
    rc=SetAttrObject((Qlg_Path_Name_T *)&pns2,
                     BufferArea,
                     buffer_size);
    if (rc != 0)
    {
      rc = errno;               /* return errno from SetAttrObject */
      printf("Qp0lSetAttr() for %s failed with %i.\n",pns2.pn,rc);
    }
  }                             /* end check GetAttrObject rc      */
  else                          /* GetAttrObject failed            */
  {
    rc = errno;                 /* return errno from GetAttrObject */
    printf("Qp0lGetAttr() for %s failed with %s.\n",pns.pn,rc);
  }
  return(rc);
}                               /* end main                        */
```

API introduced: V4R4

Qp0lUnlink()--Remove Link to File

Syntax

```
#include <Qp0lstdi.h>

int Qp0lUnlink(Qlg_Path_Name_T *Path_Name);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes on open() API.

The **Qp0lUnlink()** function, similar to the **unlink()** function, removes a directory entry that refers to a file. **Qp0lUnlink()**differs from **unlink()** in that the Path_Name parameter is a pointer to a Qlg_Path_Name_T structure instead of a pointer to a character string.

For a discussion of the authorities required, return values, and related information, see unlink()--Remove Link to File.

Parameters

Path_Name

(Input) The path name of the object to be unlinked. This path name is in the Qlg_Path_Name_T format. For more information on this structure, see Path Name Format.

Related Information

- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- unlink()--Remove Link to File
- link()--Create Link to File
- open()--Open File
- close()--Close File or Socket Descriptor
- rmdir()--Remove Directory

Example

See Code disclaimer information for information pertaining to code examples.

The following example removes a link to a file: This program was stored in a source file with CCSID 37, so the constant string "newfile" will be compiled in coded character set identifier (CCSID) 37. Therefore, the country or region and language specified are United States English, and the CCSID specified is 37.

```
#include <fcntl.h>
#include <stdio.h>
#include <Qp0lstdi.h>

main() {
  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  const char Path_Name_Del_const[2] = "/";

  struct pnstruct
    {
     Qlg_Path_Name_T  qlg_struct;
     char             pn[7];
    };
  struct pnstruct pns;
  struct pnstruct *pns_ptr = NULL;

  char fn[]="unlink.file";

  memset((void*)&pns, 0x00, sizeof(struct pnstruct));
  pns.qlg_struct.CCSID = 37;
  memcpy(pns.qlg_struct.Country_ID,US_const,2);
  memcpy(pns.qlg_struct.Language_ID,Language_const,3);;
  pns.qlg_struct.Path_Type = 0;
  pns.qlg_struct.Path_Length = sizeof(fn)-1;
  memcpy(pns.qlg_struct.Path_Name_Delimiter,
         Path_Name_Del_const,1);
  memcpy(pns.pn,fn,sizeof(fn));
  memset((void *)&Attr_types_ptr, 0x00,
         sizeof(struct attrStruct));
  pns_ptr = &pns;

  if (Qp0lUnlink((Qlg_Path_Name_T *)&pns) != 0)
      perror("Qp0lunlink() error");
}
```

API introduced: V4R4

Qp0zPipe()--Create Interprocess Channel with Sockets

Syntax

```
#include <spawn.h>

int Qp0zPipe(int fildes[2]);
```

Service Program Name: QP0ZSPWN

Default Public Authority: *USE

Threadsafe: Yes

The **Qp0zPipe()** function creates a data pipe that can be used by two processes. One end of the pipe is represented by the file descriptor returned in *fildes[0]*. The other end of the pipe is represented by the file descriptor returned in *fildes[1]*. Data that is written to one end of the pipe can be read from the other end of the pipe in a first-in-first-out basis. Both ends of the pipe are open for reading and writing.

The **Qp0zPipe()** function is often used with the **spawn()** function to allow the parent and child processes to send data to each other.

Parameters

fildes[2]
: (Input) An integer array of size 2 that will contain the pipe descriptors.

Authorities

None.

Return Value

0 **Qp0zPipe()** was successful.

-1 **Qp0zPipe()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **Qp0zPipe()** 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.

[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).

[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.

[ENOBUFS]
: There is not enough buffer space for the requested operation.

[EOPNOTSUPP]
: Operation not supported.

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

[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.

Usage Notes

The OS/400 implementation of the **Qp0zPipe()**function is based on sockets rather than pipes and, therefore, uses socket descriptors. There are several differences:

1. After calling the **fstat()** function using one of the file descriptors returned on a Qp0zPipe() call, when the st_mode from the stat structure is passed to the S_ISFIFO() macro, the return value indicates FALSE. When the st_mode from the stat structure is passed to S_ISSOCK(), the return value indicates TRUE.
2. The file descriptors returned **on a Qp0zPipe()** call can be used with the **send()**, **recv()**, **sendto()**, **recvfrom()**, **sendmsg()**, and **recvmsg()** functions.

If you want to use the traditional implementation of pipes, in which the descriptors returned are pipe descriptors instead of socket descriptors, use the **pipe()** function.

Related Information

- The <**spawn.h**> file (see Header Files for UNIX-Type Functions)
- fstat()--Get File Information by Descriptor
- pipe()--Create an Interprocess Channel
- spawn()--Spawn Process
- socketpair()--Create a Pair of Sockets
- stat()--Get File Information

API introduced: V4R1

»qsygetgroups()--Get Supplemental Group IDs

Syntax

```
#include <qsysetid.h>

int qsygetgroups(int gidsetsize, gid_t grouplist[])
```

Threadsafe: No

If the *gidsetsize* argument is zero, **qsygetgroups()** returns the number of supplemental group IDs associated with the calling thread without modifying the array pointed to by the *grouplist* argument. Otherwise, **qsygetgroups()** fills in the array *grouplist* with the 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 supplementary group IDs.

Authorities

No authorization is required.

Return Value

0 or > 0	**qsygetgroups()** was successful. If the *gidsetsize* argument is 0, the number of supplementary group IDs is returned. If *gidsetsize* is greater than 0, the array *grouplist* is filled with the supplementary group IDs of the calling thread and the return value represents the actual number of group IDs stored.
-1	**qsygetgroups()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **qsygetgroups()** 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.

API introduced: V5R2

---

qsysetegid()--Set Effective Group ID

Syntax

```
#include <qsysetid.h>

int qsysetegid(gid_t gid);
```

Threadsafe: Yes

If *gid* is equal to either the real, effective, saved group ID, or one of the groups in the supplemental group list, **qsysetegid()** sets the effective group ID to *gid*.

If *gid* is not equal to any of the current groups, but the thread has *USE authority to the user profile associated with the *gid*, **qsysetegid()** sets the effective group ID to *gid*.

Job scoped locks with a lock state of *SHRRD are held on the user profiles associated with the real user ID, effective user ID, saved user ID, real group ID, effective group ID, saved group ID, and all of the supplemental groups.

Parameters

gid

(Input) Group ID.

This field must contain one of the following values:

0

There is no effective group ID.

1 to 4294967294

The group ID value for the set operation.

Authorities and Locks

User profile associated with *uid* authority

*USE authority is required to the user profile associated with *gid* if *gid* is not equal to the real, effective, saved group IDs or one of the groups in the supplemental group list.

User profile associated with *uid* lock

*SHRRD

Return Value

0

qsysetegid() was successful.

-1

qsysetegid() was not successful. *errno* is set to indicate the error.

Error Conditions

If **qsysetegid()** is not successful, *errno* indicates one of the following errors.

[EAGAIN]

User profile associated with the *gid* is locked. Try again.

[EINVAL]

The value of the *gid* argument is invalid. Following are possible reasons:

- Out of range.
- Not associated with a user profile.

[EDAMAGE]

The user profile associated with the *gid* or an internal system object is damaged.

[ENOTSUP]

Operation not supported. The current effective user profile specifies OWNER(*GRPPRF), but the group profile associated with this *gid* is not equal to the user profile's first group and the user's first group is not in the list of supplemental groups.

[EPERM]

Operation not permitted. Following are possible reasons:

- The thread does not have *USE authority to the user profile associated with the *gid* and the *gid* to be set is not the same as the real, effective, saved group IDs or any of the supplemental groups.
- *gid* cannot be set to 0 if there are supplemental groups.

[EUNKNOWN]

An unknown error has occurred. Check the joblog for error messages.

---

qsyseteuid()--Set Effective User ID

Syntax

```
#include <qsysetid.h>

int qsyseteuid(uid_t uid);
```

Threadsafe: Yes

If *uid* is equal to the real, effective, or saved user ID, **qsyseteuid()** sets the effective user ID to *uid*.

If *uid* is not equal to the real, effective, or saved user ID, but the thread has *USE authority to the user profile associated with *uid*, **qsyseteuid()** sets the effective user ID to *uid*.

Job scoped locks with a lock state of *SHRRD are held on the user profiles associated with the real user ID, effective user ID, saved user ID, real group ID, effective group ID, saved group ID, and all of the supplemental groups.

Parameters

uid

(Input) User ID.

This field must contain one of the following values:

0 to 4294967294

The user ID value for the set operation.

Authorities and Locks

User profile associated with *uid* authority

*USE authority is required to the user profile associated with *uid* if *uid* is not equal to the real, effective or saved user IDs.

User profile associated with *uid* lock

*SHRRD

Return Value

0

qsyseteuid() was successful.

-1

qsyseteuid() was not successful. *errno* is set to indicate the error.

Error Conditions

If **qsyseteuid()** is not successful, *errno* indicates one of the following errors.

[EAGAIN]

User profile associated with the *uid* is locked. Try again.

[EDAMAGE]

The user profile associated with the *uid* or an internal system object is damaged.

[EINVAL]

The value of the *uid* argument is invalid. Following are possible reasons:

- Out of range.
- Not associated with a user profile.

[ENOTSUP]

Operation not supported. The user profile associated with this *uid* specifies OWNER(*GRPPRF), but the user profile's first group is not the current effective group, nor is it in the list of supplemental groups.

[EPERM]

Operation not permitted. The thread does not have *USE authority to the user profile and the *uid* to be set is not the same as the real, effective, or saved user IDs.

[EUNKNOWN]

An unknown error has occurred. Check the joblog for error messages.

---

qsysetgid()--Set Group ID

Syntax

```
#include <qsysetid.h>

int qsysetgid(gid_t gid);
```

Threadsafe: Yes

If the thread has *ALLOBJ special authority, **qsysetgid()** sets the real, effective and saved groups to *gid*.

If the thread does not have *ALLOBJ special authority, but *gid* is equal to the real, effective or saved group IDs, the **qsysetgid()** function sets the effective group ID to *gid*. The real group and saved group IDs remain unchanged.

Any supplementary group IDs of the calling thread remain unchanged.

Job scoped locks with a lock state of *SHRRD are held on the user profiles associated with the real user ID, effective user ID, saved user ID, real group ID, effective group ID, saved group ID, and all of the supplemental groups.

Parameters

gid

(Input) Group ID.

This field must contain one of the following values:

0

There is no group ID. The effective group ID can be set to 0 only if there are no supplemental groups.

1 to 4294967294

The group ID value for the set operation.

Authorities and Locks

***ALLOBJ special authority**

*ALLOBJ special authority is required if *gid* is not equal to the real, effective or saved group ID.

User profile associated with *gid* lock

*SHRRD

Return Value

0

qsysetgid() was successful.

-1

qsysetgid() was not successful. *errno* is set to indicate the error.

Error Conditions

If **qsysetgid()** is not successful, *errno* indicates one of the following errors.

[EAGAIN]

User profile associated with the *gid* is locked. Try again.

[EDAMAGE]

The user profile associated with the *gid* or an internal system object is damaged.

[EINVAL]

The value of the *gid* argument is invalid. Following are possible reasons:

- Out of range.
- Not associated with a user profile.

[ENOTSUP]

Operation not supported. The current effective user profile specifies OWNER(*GRPPRF), but the group profile associated with this *gid* is not equal to the user profile's first group and the user's first group is not in the list of supplemental groups.

[EPERM]

Operation not permitted. Following are possible reasons:

- The thread does not have *ALLOBJ special authority and *gid* is not the same as the real, effective or saved group ID.
- Tried to set effective group ID to 0 when there are supplemental groups.

[EUNKNOWN]

An unknown error has occurred. Check the joblog for error messages.

»qsysetgroups()--Set Supplemental Group IDs

Syntax

```
#include <qsysetid.h>

int qsysetgroups(int gidsetsize, gid_t grouplist[])
```

Threadsafe: No

The qsysetgroups API sets the supplementary group IDs of the calling thread. The qsysetgroups API cannot set more than (NGROUPS_MAX-1) groups in the group set.

Parameters

gidsetsize

(Input) The number of elements in the supplied array *grouplist*.

grouplist

(Input) The supplementary group IDs.

Authorities and locks

User profile associated with *gid* Authority

*USE authority is required to the user profile associated with each *gid* in the group list if the *gid* is not equal to the current thread's real, effective, or saved group IDs or one of the groups in the current thread's supplemental group list.

User profile associated with each *gid* Lock

*SHRRD

Return Value

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

Error Conditions

If **qsysetgroups()** 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]

User profile associated with a *gid* is locked. Try again.

[EDAMAGE]

The user profile associated with a gid or an internal system object is damaged.

[EINVAL]

One of the GID values in the *grouplist* argument is not valid. Following are possible reasons:

- Out of range.
- Not associated with a user profile.
- gidsetsize too large.

[ENOTSUP]

Operation not supported. The current effective user profile specifies OWNER(*GRPPRF), but the user's first group is not equal to the current effective group profile and the user's first group is not in this list of supplemental groups.

[EPERM]

Operation not permitted. Following are possible reasons:

- The thread does not have *USE authority to the user profile associated with the *GID* and the *GID* to be set is not the same as the real, effective, saved group IDs or any of the supplemental groups.
- Supplemental groups cannot be set if effective GID is 0.

[EUNKNOWN]

An unknown error has occurred. Check the joblog for error messages.

API introduced: V5R2

---

qsysetregid()--Set Real and Effective Group IDs

Syntax

```
#include <qsysetids.h>

int qsysetregid(gid_t rgid, gid_t egid);
```

Threadsafe: Yes

The **qsysetregid()** function is used to set the real and effective group IDs. The real and effective group IDs may be set to different values in the same call.

A thread with *ALLOBJ special authority can set the real group ID and the effective group ID to any valid value.

A thread without *ALLOBJ special authority can only set the real group ID to the saved group ID. A thread without *ALLOBJ special authority can only set the effective group ID to the saved group ID or the real group ID.

Any supplemental group IDs remain unchanged.

Job scoped locks with a lock state of *SHRRD are held on the user profiles associated with the real user ID, effective user ID, saved user ID, real group ID, effective group ID, saved group ID, and all of the supplemental groups.

Parameters

real gid

(Input) Group ID.

This field must contain one of the following values:

0

There is no real group ID.

1 to 4294967294

The group ID value for the set operation.

4294967295

The real group ID does not change. This value is the same as X'FFFFFFFF' or -1 in languages that do not support unsigned integers.

effective gid

(Input) Group ID.

This field must contain one of the following values:

0

There is no effective group ID.

1 to 4294967294

The group ID value for the set operation.

4294967295

The effective group ID does not change. This value is the same as X'FFFFFFFF' or -1 in languages that do not support unsigned integers.

Authorities and Locks

***ALLOBJ special authority**

*ALLOBJ special authority is required to change the real group ID if *rgid* is not equal to the saved group ID. *ALLOBJ special authority is required to set the effective group ID if the *egid* is not equal to the real group ID or the saved group ID.

User profile associated with *rgid* lock

*SHRRD

User profile associated with *egid* lock

*SHRRD

Return Value

0

qsysetregid() was successful.

-1

qsysetregid() was not successful. The *errno* is set to indicate the error.

Error Conditions

If **qsysetregid()** is not successful, *errno* indicates one of the following errors.

[EAGAIN]

User profile associated with the rgid or rgid is locked. Try again.

[EDAMAGE]

The user profile associated with one of the gids or an internal system object is damaged.

[EINVAL]

The value of the gid argument is invalid. Following are possible reasons:

- Out of range.
- Not associated with a user profile.

[ENOTSUP]

Operation not supported. The current effective user profile specifies OWNER(*GRPPRF), but the group profile associated with this *gid* is not equal to the user profile's first group and the user's first group is not in the list of supplemental groups.

[EPERM]

Operation not permitted. Following are possible reasons:

- The thread does not have *ALLOBJ special authority and a change other than changing the real group ID to the saved group ID, or changing the effective group ID to the real group ID or the saved group ID was requested.
- Tried to set effective group ID to 0 when there are supplemental groups.

[EUNKNOWN]

An unknown error has occurred. Check the joblog for error messages.

Top | UNIX-Type APIs | APIs by category

qsysetreuid()--Set Real and Effective User IDs

Syntax

```
int qsysetreuid(uid_t ruid, uid_t euid);
```

Threadsafe: Yes

The **qsysetreuid()** function sets the real and effective user IDs to the values specified by *ruid* and *euid*.

A thread with *ALLOBJ special authority can set either ID to any value.

A thread without *ALLOBJ special authority can only set the effective user ID if the *euid* argument is equal to the real, effective, or saved user ID.

Job scoped locks with a lock state of *SHRRD are held on the user profiles associated with the real user ID, effective user ID, saved user ID, real group ID, effective group ID, saved group ID, and all of the supplemental groups.

Parameters

real uid

(Input) User ID.

This field must contain one of the following values:

0 to 4294967294

The user ID value for the set operation.

4294967295

The real user ID does not change. This value is the same as X'FFFFFFFF' or -1 in languages that do not support unsigned integers.

effective uid

(Input) User ID.

This field must contain one of the following values:

0 to 4294967294

The user ID value for the set operation.

4294967295

The effective user ID does not change. This value is the same as X'FFFFFFFF' or -1 in languages that do not support unsigned integers.

Authorities and Locks

***ALLOBJ special authority**

*ALLOBJ special authority is required to change the real user ID. *ALLOBJ special authorty is required to change the effective user ID if the *euid* is not equal to the real, effective, or saved user ID.

User profile associated with *euid* lock

*SHRRD

User profile associated with *ruid* lock

*SHRRD

Return Value

0

qsysetreuid() was successful.

-1

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

Error Conditions

If **qsysetreuid()** is not successful, *errno* indicates one of the following errors.

[EAGAIN]

User profile associated with *ruid* or *euid* is locked. Try again.

[EDAMAGE]

The user profile associated with *ruid* or *euid* or an internal system object is damaged.

[EINVAL]

The value of the *ruid* or *euid* argument is invalid. Following are possible reasons:

- Out of range.
- Not associated with a user profile.

[ENOTSUP]

Operation not supported. The user profile associated with this *uid* specifies OWNER(*GRPPRF), but the user profile's first group is not the current effective group, nor is it in the list of supplemental groups.

[EPERM]

Operation not permitted. The current thread does not have *ALLOBJ special authority, and either an attempt was made to change the effective user ID to a value other than the real user ID or the saved set-user-ID or an an attempt was made to change the real user ID.

[EUNKNOWN]

An unknown error has occurred. Check the joblog for error messages.

---

qsysetuid()--Set User ID

Syntax

```
#include <qsysetid.h>

int qsysetuid(uid_t uid);
```

Threadsafe: Yes

If the thread has *ALLOBJ special authority, **qsysetuid**() sets the real, effective, and saved user ID to *uid*.

If the thread does not have *ALLOBJ special authority, but *uid* is equal to the real, effective or saved user ID, **qsysetuid**() sets the effective user ID to *uid*. The real and saved user IDs remain unchanged.

Job scoped locks with a lock state of *SHRRD are held on the user profiles associated with the real user ID, effective user ID, saved user ID, real group ID, effective group ID, saved group ID, and all of the supplemental groups.

Parameters

uid

(Input) User ID.

This field must contain one of the following values:

0 to 4294967294

The user ID value for the set operation.

Authorities and Locks

***ALLOBJ special authority**

*ALLOBJ special authority is required if *uid* is not equal to the real, effective, or saved user ID.

User profile associated with *uid* lock

*SHRRD

Return Value

0

qsysetuid() was successful.

-1

qsysetuid() was not successful. *errno* is set to indicate the error.

Error Conditions

If **qsysetuid()** is not successful, *errno* indicates one of the following errors.

[EAGAIN]

User profile associated with the *uid* is locked. Try again.

[EDAMAGE]

The user profile associated with the *uid* or an internal system object is damaged.

[EINVAL]

The value of the *uid* is invalid. Following are possible reasons:

- Out of range.
- Not associated with a user profile.

[ENOTSUP]

Operation not supported. The user profile associated with this *uid* specifies OWNER(*GRPPRF), but the user profile's first group is not the current effective group, nor is it in the list of supplemental groups.

[EPERM]

Operation not permitted. The thread does not have *ALLOBJ special authority and *uid* is not the same as the real, effective or saved user ID.

[EUNKNOWN]

An unknown error has occurred. Check the joblog for error messages.

Retrieve Network File System Export Entries (QZNFRTVE) API

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Length of receiver variable in bytes	Input	Binary(4)
3	Returned records feedback information	Output	Char(16)
4	Format name	Input	Char(8)
5	Object path name	Input	Char(*)
6	Length of object path name in bytes	Input	Binary(4)
7	CCSID of object path name given	Input	Binary(4)
8	Desired CCSID of the object path names returned	Input	Binary(4)
9	Handle	Input	Binary(4)
10	Error code	I/O	Char(*)

Threadsafe: No

The Retrieve Network File System Export Entries (QZNFRTVE) API returns the list of Network File System (NFS) export entries for objects currently exported to NFS clients or for objects referenced in the /etc/exports file.

Authorities and Locks

- The user must have execute (*X) data authority to the /etc directory (if it exists).
- The user must have read (*R) data authority to the /etc/exports file (if it exists).

Note: Adopted authority is not used.

Usage Notes

If none of the required parameters are passed to this API, then all of the entries that are currently exported will be returned to the joblog by messages (CPIB41A). If there are no entries currently exported, then message CPIB41B will be returned.

Required Parameter Group

The following parameters are required.

Receiver variable

OUTPUT; CHAR(*)

The receiver variable that receives the information requested. The API returns only data that the area can hold.

Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable provided. The length of the receiver variable parameter may be specified up to the size of the receiver variable area specified by the user program.

No partial entries will be returned. If the length of the receiver variable is less than what is required by the format selected, then an error is returned (CPFB419) and the size required will be indicated in the feedback structure.

Returned records feedback information

OUTPUT; CHAR(16)

Information about the entries that are returned in the receiver variable.

For a detailed description of this format, see Format of Returned Records Feedback Information.

Format name

INPUT; CHAR(8)

The name of the format that is used to retrieve NFS export entries.

You can specify one of the following formats:

EXPE0100

Returns information about export entries that are currently exported. These are sometimes called temporary exports. For a detailed description of this format, see EXPE0100 and EXPE0200 format.

EXPE0200

Returns information about export entries that are in the /etc/exports file. These are sometimes called permanent exports. For a detailed description of this format, see EXPE0100 and EXPE0200 format.

Object path name

INPUT; CHAR(*)

The object path name at which to start listing NFS export entries. Possible values follow:

**FIRST*

NFS export entries are returned starting with the first object path name in the NFS export entry list.

**HANDLE*

NFS export entries are returned starting with the object path name that corresponds to the specified handle.

When the receiver variable is not large enough to hold all of the entries in the NFS export entry list, the API returns a non-zero handle in the returned records feedback information parameter. This handle can be used on a subsequent call to the API to continue retrieving NFS export entries with the next object path name in the NFS export entry list.

There is no implied order to the export entries that are returned. While no sorting or

sequencing has been done on the returned entries, a complete list will eventually be returned if the *HANDLE option is used.

Object path name

The NFS export entry for the specified object path name is returned.

Length of object path name

INPUT; BINARY(4)

The length of the object path name in bytes. If one of the special values is given for the object path name, then the length should be given for that special value.

CCSID of object path name given

INPUT; BINARY(4)

The CCSID of the object path name given as an input parameter. Possible values follow:

0

The current Default Job CCSID should be used.

value

A valid CCSID number.

Desired CCSID of object the path names returned.

INPUT; BINARY(4)

The Desired CCSID of the object path names returned. The output structure will contain the actual CCSID of the returned object path names. This will match the Desired CCSID given as input, if possible. Possible values follow:

0

The current Default Job CCSID should be used.

value

A valid CCSID number.

Handle of starting object path name

INPUT; BINARY(4)

The handle returned from a previous call to the QZNFRTVE API.

This parameter should be 0 if *HANDLE was NOT specified for the object path name parameter.

Error code

I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see .

Receiver Variable Description

The following table describes the order and format of the data returned in the receiver variable. For a detailed description of each field, see Field Descriptions.

EXPE0100 and EXPE0200 format

This structure is used to return NFS export information for a single object path name for both the EXPE0100 and the EXPE0200 formats.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Length of entry
4	4	BINARY(4)	Displacement to object path name
8	8	BINARY(4)	Length of object path name
12	C	BINARY(4)	CCSID of object path name
16	10	BINARY(4)	Read-only flag
20	14	BINARY(4)	NOSUID flag
24	18	BINARY(4)	Displacement to read-write host names
28	1C	BINARY(4)	Number of read-write host names
32	20	BINARY(4)	Displacement to root host names
36	24	BINARY(4)	Number of root host names
40	28	BINARY(4)	Displacement to access host names
44	2C	BINARY(4)	Number of access host names
48	30	BINARY(4)	Displacement to host options
52	34	BINARY(4)	Number of host options
56	38	BINARY(4)	Anonymous user ID
60	3C	CHAR(10)	Anonymous User Profile
*	*	CHAR(*)	Object path name
These fields repeat for each host name in the read-write access list.		BINARY(4)	Length of host name entry
		BINARY(4)	Length of host name
		CHAR(*)	Host name
These fields repeat for each host name in the root access list.		BINARY(4)	Length of host name entry
		BINARY(4)	Length of host name
		CHAR(*)	Host name
These fields repeat for each host name in the access list.		BINARY(4)	Length of host name entry
		BINARY(4)	Length of host name
		CHAR(*)	Host name
These fields repeat for each host name in the host options list.		BINARY(4)	Length of host name options entry
		BINARY(4)	Network data file CCSID
		BINARY(4)	Network path name CCSID
		BINARY(4)	Write mode flag
		BINARY(4)	Length of host name

	CHAR(*)	Host name

Returned Records Feedback Information Description

The following table describes the order and format of the data returned in the returned records feedback information parameter. For a detailed description of each field, see Field Descriptions.

Format of Returned Records Feedback Information

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Number of NFS export entries
12	C	BINARY(4)	Handle

Field Descriptions

Anonymous User ID. The user ID used as the effective user ID for requests from unknown users. Hex value 0xFFFFFFFF (a value of -1 if this were a signed integer) indicates requests from unknown users are not allowed.

Anonymous User Profile. This is the OS/400 User Profile name that is associated with the Anonymous User ID returned. If the Anonymous User ID has the special value of hex value 0xFFFFFFFF (a value of -1 if this were a signed integer), then the Anonymous User Profile will be set to the special value of *NONE.

Bytes available. The number of bytes of data available to be returned to the user in the receiver variable. If all data is returned, bytes available is the same as the number of bytes returned. If the receiver variable was not large enough to contain all of the data, this value is estimated based on the total number of entries in the NFS export entry list that could be returned.

Bytes returned. The number of bytes of data returned to the user in the receiver variable.

CCSID of object path name. The CCSID of the object path name.

Object path name. The path name of the object for which export information is to be returned.

Displacement to access host names. The offset (in bytes) from the beginning of the NFS export entry to the host names in the access list.

Displacement to host options. The offset (in bytes) from the beginning of the NFS export entry to the host options list.

Displacement to object path name. The offset (in bytes) from the beginning of the NFS export entry to the object path name.

Displacement to read-write host names. The offset (in bytes) from the beginning of the NFS export entry to the host names in the read-write access list.

Displacement to root host names. The offset (in bytes) from the beginning of the NFS export entry to the host names in the root access list.

Handle. The handle to be used on a subsequent call to the API to continue retrieving NFS export entries with the next object path name in the NFS export entry list. 0 indicates all remaining NFS export entries have been returned.

Host name. The host name.

Length of entry. The length (in bytes) of the current NFS export entry. The length can be used to access the next entry.

Length of host name. The length (in bytes) of the host name.

Length of host name entry. The length (in bytes) of this host name entry.

Length of host name options entry. The length (in bytes) of this host name options entry.

Length of object path name. The length (in bytes) of the object path name.

Network data file CCSID. The CCSID used for data of the files sent to and received from the specified host name.

Network path name CCSID. The CCSID used for path name components of the files sent to and received from the specified host name.

NOSUID flag. Whether an attempt by the client to enable bits other than the permission bits will be ignored. Possible values follow:

0

An attempt to set bits other than the permission bits will be carried out.

1

An attempt to set bits other than the permission bits will be ignored.

Number of access host names. The number of host names in the access list.

Number of host options. The number of entries in the host options list.

Number of NFS export entries. The number of complete entries returned in the list of NFS export entries. A value of zero is returned if the list is empty relative to the requested starting position.

Number of read-write host names. The number of host names in the read-write access list.

Number of root host names. The number of host names in the root access list.

Read-only flag. Whether the object is exported allowing only read access. Possible values follow:

0

The object is exported allowing read-write access for all client hosts that are not specifically indicated to have read-only access.

1

The object is exported allowing read-only access for all client hosts that are not specifically indicated to have read-write access.

Write mode flag. Whether write requests are handled synchronously or asynchronously. Synchronously means that data will be written to disk immediately. Asynchronously does not guarantee that data is written

to disk immediately, and can be used to improve server performance. Possible values follow:

0

Write requests are performed synchronously.

1

Write requests are performed asynchronously.

Error Messages

CPE3418 E

Possible APAR condition or hardware failure.

CPF3C90 E

Literal value cannot be changed.

CPF3CF2 E

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

CPF9872 E

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

CPFA0D4 E

File system error occurred.

---

read()--Read from Descriptor

Syntax

```
#include <unistd.h>

ssize_t read(int file_descriptor,
             void *buf, size_t nbyte);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

From the file or socket indicated by *file_descriptor*, the **read()** function reads *nbyte* bytes of input into the memory area indicated by *buf*. If *nbyte* is zero, **read()** returns a value of zero without attempting any other action.

If *file_descriptor* refers to a "regular file" (a stream file that can support positioning the file offset) or any other type of file on which the job can do an **lseek()** operation, **read()** begins reading at the file offset associated with *file_descriptor*. A successful **read()** changes the file offset by the number of bytes read.

If **read()** is successful and *nbyte* is greater than zero, the access time for the file is updated.

read() is not supported for directories.

If *file_descriptor* refers to a descriptor obtained using the **open**() function with O_TEXTDATA specified, the data is read from the file assuming it is in textual form. The maximum number of bytes on a single read that can be supported for text data is 2,147,483,408 (2GB - 240) bytes. The data is converted from the code page of the file to the code page of the application, job, or system as follows:

- When reading from a true stream file, any line-formatting characters (such as carriage return, tab, and end-of-file) are just converted from one code page to another.
- When reading from record files that are being used as stream files, end-of-line characters are added to the end of the data in each record.

There are some important considerations when the file is open for text conversion and the CCSIDs involved are not strictly single-byte:

- The **read()** will return the exact number of bytes requested. For some CCSIDs, this may mean that partial characters are returned at the end of the user buffer. In this case, the remainder of the character has been read from the file and internally buffered. The next consecutive read() will begin with the remainder of the partial character. However, if an **lseek()** is performed, the buffered data will be discarded. See lseek()--Set File Read/Write Offset for more information.
- Because of the above consideration and because of the possible expansion or contraction of converted data, applications using the O_CCSID flag should avoid assumptions about data size and the current file offset. 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 50. In order to read the whole file, the application might have to read 200 bytes or more, depending on the CCSIDs involved.

If O_TEXTDATA was not specified on the **open**(), the data is read from the file without conversion. The application is responsible for handling the data.

In the QSYS.LIB »and independent ASP QSYS.LIB file systems, «most end-of-file characters are symbolic; that is, they are stored outside the member. When reading:

- If O_TEXTDATA is specified, both symbolic and nonsymbolic end-of-file characters can be seen.
- If O_TEXTDATA is not specified (binary mode), only nonsymbolic end-of-file characters can be seen.

See the *Usage Notes* for write()--Write to Descriptor.

When *file_descriptor* refers to a socket, the **read()** function reads from the socket identified by the socket descriptor.

When attempting to read from an empty pipe or FIFO:

- If no job has the pipe or FIFO open for writing, **read()** return 0 to indicate end-of-file.
- If some job has the pipe or FIFO open for writing and O_NONBLOCK was specified, **read()** will fail and errno will be set to [EAGAIN].
- If some job has the pipe or FIFO open for writing and O_NONBLOCK was not specified, **read()** will block the calling thread until some data is written or until the pipe or FIFO is closed by all jobs that had the pipe or FIFO open for writing.

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.

Authorities

No authorization is required.

Return Value

value

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

-1

read() was not successful. The *errno* global variable is set to indicate the error. If the value of *nbyte* is greater than SSIZE_MAX, **read()** sets *errno* to [EINVAL].

Error Conditions

If **read()** 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.

This may occur if *file_descriptor* refers to a socket and the socket is using a connection-oriented transport service, and a *connect()* was previously completed. The thread, however, does not have the appropriate privileges to the objects that were needed to establish a connection. For example, the *connect()* required the use of an APPC device that the thread was not authorized to.

[EAGAIN] Operation would have caused the process to be suspended.

If *file_descriptor* refers to a pipe or FIFO that has its O_NONBLOCK flag set, this error occurs if the **read()** would have blocked the calling thread.

[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 **read** 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 *file_descriptor* refers to a socket that is using a connectionless transport service, is not a socket of type SOCK_RAW, and is not bound to an address.

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.

[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.«

[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.

When the descriptor refers to a socket, *errno* could indicate one of the following errors:

[ECONNABORTED] Connection ended abnormally.

This error code indicates that the transport provider ended the connection abnormally because of one of the following:

- The retransmission limit has been reached for data that was being sent on the socket.
- A protocol error was detected.

[ECONNREFUSED] The destination socket refused an attempted connect operation.

[ECONNRESET] A connection with a remote socket was reset by that socket.

[EINTR] Interrupted function call.

[ENOTCONN] Requested operation requires a connection.

This error code is returned only on sockets that use a connection-oriented transport service.

[ETIMEDOUT] A remote host did not respond within the timeout period.

A non-blocking **connect()** was previously completed that resulted in the connection timing out. No connection is established. This error code is returned only on sockets that use a connection-oriented transport service.

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

[EWOULDBLOCK] Operation would have caused the process to be suspended.

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. 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 **read()** 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 sockets that use a connection-oriented transport service (for example, sockets with a type of SOCK_STREAM), a return value of zero indicates one of the following:
   - The partner program has issued a **close()** for the socket.
   - The partner program has issued a **shutdown()** to disable writing to the socket.
   - The connection is broken and the error was returned on a previously issued socket function.
   - A **shutdown()** to disable reading was previously done on the socket.

7. The following applies to sockets that use a connectionless transport service (for example, a socket with a type of SOCK_DGRAM).
   - If a **connect()** has been issued previously, then data can be received only from the address specified in the previous **connect()**.
   - The address from which data is received is discarded, since the **read()** has no address parameter.
   - The entire message must be read in a single read operation. If the size of the message is too large to fit in the user supplied buffer, the remaining bytes of the message are discarded.
   - A returned value of zero indicates one of the following:
     - The partner program has sent a NULL message (a datagram with no user data).
     - A **shutdown()** to disable reading was previously done on the socket.
     - The buffer length specified was zero.

8. For file systems that do not support large files, **read()** 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, **read()** will return [EOVERFLOW] if the starting offset exceeds 2GB minus 2 bytes and the file was not opened for large file access.

9. 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.

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
- 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)
- 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, rc;
  char buf[]="Test text";

  if ((file_descriptor = creat("test.output", S_IWUSR))!= 0)
     perror("creat() error");
  else {
    if (-1==(rc=write(file_descriptor, buf, sizof(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 = read(file_descriptor, buf, sizeof(buf)-1));
    buf[ret] = 0x00;
    printf("block read: \n<%s>\", buf);
    if (close(file_descriptor)!= 0)
       perror("close() error");
  }
```

```
  if (unlink("test.output")!= 0)
     perror("unlink() error");
}
```

Output:

```
block read:
<Test text>
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

readdir()--Read Directory Entry

Syntax

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

struct dirent *readdir(DIR *dirp);
```

Threadsafe: No; see Usage Notes.

The **readdir()** function returns a pointer to a dirent structure describing the next directory entry in the directory stream associated with *dirp*.

A call to **readdir()** overwrites data produced by a previous call to **readdir()** on the same directory stream. Calls for different directory streams do not overwrite the data of each other.

If the call to **readdir()** actually reads the directory, the access time of the directory is updated.

readdir() performs translation if necessary to convert the directory entry name into the CCSID (coded character set identifier) of the job at the time of the call to **opendir()**.

Parameters

dirp

(Input) A pointer to a DIR that refers to theopen directory stream to be read. This pointer is returned by **opendir()** (see opendir()--Open Directory).

See QlgReaddir()--Read Directory Entry for a description and an example of supplying the *dirp* in any CCSID, using a dirent_lg structure.

A dirent structure has the following contents:

char	d_reserved1[16]	Reserved.
unsigned int	d_fileno_gen_id	The generation ID associated with the file ID.
ino_t	d_fileno	The file ID of the file. This number uniquely identifies the object within a file system.
unsigned int	d_reclen	The length of the directory entry in bytes.
int	d_reserved3	Reserved.
char	d_reserved4[6]	Reserved.
char	d_reserved5[2]	Reserved.

qlg_nls_t	d_nlsinfo	National language information about d_name. The following fields are defined: *int ccsid* CCSID of the characters in the d_name field. *char country_id[2]* Country or region identifier associated with the d_name field. *char language_id[3]* Language identifier associated with the d_name field. *char nls_reserved[3]* Reserved.
unsigned int	d_namelen	The length of the name in bytes, excluding the null terminator.
char	d_name[640]	A string that gives the name of a file in the directory. This string ends in a terminating null, and has a maximum length of {NAME_MAX} bytes, not including the terminating NULL (see pathconf()--Get Configurable Path Name Variables).

Authorities

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

Return Value

value

readdir() was successful. The value returned is a pointer to a dirent structure describing the next directory entry in the directory stream.

NULL pointer

One of the following has occurred:

- **readdir()** reached the end of the directory stream. The *errno* global variable is not changed.
- **readdir()** was not successful. The *errno* is set.

Error Conditions

If **readdir()** 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.

[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.

[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.

[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.

[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.

[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. The **readdir_r()** API should be used to read a directory when running in a multithreaded job.
2. Save the data from **readdir()**, if required, before calling **closedir()**, because **closedir()** frees the data.
3. If the *dirp* argument passed to **readdir()** does not refer to an open directory stream, **readdir()** returns the [EBADF] error.
4. **readdir()** buffers multiple directory entries to improve performance. This means the directory is not actually read on each call to **readdir()**. As a result, files that are added to the directory after

   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. **readdir()** also returns directory entries for dot (.) and dot-dot (..) subdirectories.
6. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   Calls to **readdir()** that update the access time of the directory use the normal rules that apply to libraries and database files. At most, the access time is updated once per day.

7. QDLS File System Differences

   The access time of the directory is updated on **opendir()**. The access time is not affected by **readdir()**.

   When objects in QDLS are accessed, the country or region ID and language ID of the directory entry name are always set to the country or region ID and language ID of the system.

   When a **readdir()** operation specifies the /QDLS directory, the user must have *USE authority to each child object of the /QDLS directory (that is, *USE authority to each object immediately below QDLS in the directory hierarchy). A directory entry is returned only for those objects for which the user has *USE authority. If the **readdir()** operation specifies a directory below QDLS, a directory entry is returned for all objects, even if the user does not have *USE authority for some of the objects.

8. QOPT File System Differences

   The access time of the directory is not updated on a **readdir()** operation.

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
- QlgReaddir()--Read Directory Entry
- rewinddir()--Reset Directory Stream to Beginning
- closedir()--Close Directory
- pathconf()--Get Configurable Path Name Variables

Example

The following example reads the contents of a root directory:

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

```
main() {
  DIR *dir;
  struct dirent *entry;

  if ((dir = opendir("/")) == NULL)
    perror("opendir() error");
  else {
    puts("contents of root:");
    while ((entry = readdir(dir)) != NULL)
      printf("  %s\n", entry->d_name);
    closedir(dir);
  }
}
```

Output:

```
contents of root:
  .
  ..
  QSYS.LIB
  QDLS
  QOpenSys
  QOPT
  home
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

readdir_r()--Read Directory Entry

Syntax

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

int readdir_r(DIR *dirp, struct dirent *entry,
              struct dirent **result);
```

Threadsafe: Conditional; see Usage Notes.

The **readdir_r()** function initializes the dirent structure that is referenced by *entry* to represent the next directory entry in the directory stream that is associated with *dirp*. The **readdir_r()** function then stores a pointer to the *entry* structure at the location referenced by *result*.

The storage pointed to by *entry* must be large enough for a dirent structure.

If the call to **readdir_r()** actually reads the directory, the access time of the directory is updated.

The **readdir_r()** function performs translation, if necessary, to convert the directory entry name into the coded character set identifier (CCSID) of the job at the time of the call to **opendir()**.

Parameters

dirp

(Input) A pointer to a DIR that refers to the open directory stream to be read. This pointer is returned by **opendir()** (see opendir()--Open Directory).

See QlgReaddir()--Read Directory Entry for a description and an example of supplying the *dirp* in any CCSID.

entry

(Output) A pointer to a dirent structure in which the directory entry is to be placed.

result

(Output) A pointer to a pointer to a dirent structure. Upon successfully reading a directory entry, this dirent pointer is set to the same value as *entry*. Upon reaching the end of the directory stream, this pointer will be set to NULL.

A dirent structure has the following contents:

char	d_reserved1[16]	Reserved.
unsigned int	d_fileno_gen_id	The generation ID associated with the file ID.
ino_t	d_fileno	The file ID of the file. This number uniquely identifies the object within a file system.
unsigned int	d_reclen	The length of the directory entry in bytes.
int	d_reserved3	Reserved.
char	d_reserved4[6]	Reserved.

char	d_reserved5[2]	Reserved.
qlg_nls_t	d_nlsinfo	National language information about d_name. The following fields are defined: *int ccsid* CCSID of the characters in the d_name field. *char country_id[2]* Country or region identifier that is associated with the d_name field. *char language_id[3]* Language identifier that is associated with the d_name field. *char nls_reserved[3]* Reserved.
unsigned int	d_namelen	The length of the name in bytes, excluding the null terminator.
char	d_name[640]	A string that gives the name of a file in the directory. This string ends in a terminating null, and has a maximum length of {NAME_MAX} bytes, not including the terminating NULL (see pathconf()--Get Configurable Path Name Variables).

Authorities

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

Return Value

0

readdir_r() was successful. The *result* parameter points to one of the following:

- A pointer to a dirent structure that describes the next directory entry in the directory stream. This will be the same value as the *entry* parameter.
- A NULL pointer. **readdir_r()** reached the end of the directory stream. The *errno* global variable is not changed.

error code

readdir_r() was not successful. This value is set to the same value as the *errno* global variable.

Error Conditions

If **readdir_r()** 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.

[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.

[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.

[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.

[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.

[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. **readdir_r()** is threadsafe only when directed to a directory in a threadsafe file system.
3. If the *dirp* argument that is passed to **readdir_r()** does not refer to an open directory stream, **readdir_r()** returns the [EBADF] error.
4. **readdir_r()** caches multiple directory entries to improve performance. This means the directory is not actually read on each call to **readdir_r()**. As a result, files that are added to the directory after **opendir()** or **rewinddir()** may not be returned on calls to **readdir_r()**, and files that are removed may still be returned on calls to **readdir_r()**.
5. **readdir_r()** also returns directory entries for dot (.) and dot-dot (..) subdirectories.
6. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   Calls to **readdir_r()** that update the access time of the directory use the normal rules that apply to libraries and database files. At most, the access time is updated once per day.

7. QDLS File System Differences

   The access time of the directory is updated on **opendir()**. The access time is not affected by **readdir_r()**.

   When objects in QDLS are accessed, the country or region ID and language ID of the directory entry name are always set to the country or region ID and language ID of the system.

   When a **readdir_r()** operation specifies the /QDLS directory, the user must have *USE authority to each object in the /QDLS directory (that is, *USE authority to each object immediately below QDLS in the directory hierarchy). A directory entry is returned only for those objects for which the user has *USE authority. If the **readdir_r()** operation specifies a directory below QDLS, a directory entry is returned for all objects, even if the user does not have *USE authority for some of the objects.

8. QOPT File System Differences

   The access time of the directory is not updated on a **readdir_r()** operation.

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
- QlgReaddir()--Read Directory Entry

- readdir_r_ts64()--Read Directory Entry
- rewinddir()--Reset Directory Stream to Beginning
- closedir()--Close Directory
- pathconf()--Get Configurable Path Name Variables

Example

The following example reads the contents of a root directory:

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

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

  if ((dir = opendir("/")) == NULL)
    perror("opendir() error");
  else {
    puts("contents of root:");
    for (return_code = readdir_r(dir, &entry, &result);
         result != NULL && return_code == 0;
         return_code = readdir_r(dir, &entry, &result))
      printf("  %s\n", entry.d_name);
    if (return_code != 0)
      perror("readdir_r() error");
    closedir(dir);
  }
}
```

Output:

```
contents of root:
  .
  ..
  QSYS.LIB
  QDLS
  QOpenSys
  QOPT
  home
```

API introduced: V3R1

---

readdir_r_ts64()--Read Directory Entry

Syntax

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

int readdir_r_ts64(DIR * __ptr64 dirp,
                    struct dirent * __ptr64 entry,
                    struct dirent * __ptr64 * __ptr64 result);
```

Service Program Name: QP0LLIBTS

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **readdir_r_ts64()** function initializes the dirent structure that is referenced by *entry* to represent the next directory entry in the directory stream that is associated with *dirp*. **readdir_r_ts64()** differs from **readdir_r()** in that it accepts 8-byte process local pointers.

For a discussion of the parameters, authorities required, return values, related information, usage notes, and an example for the **readdir_r()** API, see readdir_r()--Read Directory Entry.

API introduced: V5R1

---

readlink()--Read Value of Symbolic Link

Syntax

```
#include <unistd.h>

int readlink(const char *path, char *buf, size_t bufsiz);
```

Threadsafe: Conditional; see Usage Notes.

The **readlink()** function places the contents of the symbolic link *path* in the buffer *buf*. The size of the buffer is set by *bufsiz*. The contents of the returned buffer do not include a terminating NULL. When *bufsiz* is 0, the number of bytes contained in the symbolic link is returned and the buffer is unchanged.

If the buffer is too small to contain the contents of the symbolic link, the contents are truncated to the size of the buffer (*bufsiz*).

A successful **readlink()**, where *bufsiz* is greater than zero, sets the access time of the symbolic link.

Parameters

path

(Input) A pointer to the null-terminated path name of the symbolic link.

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 QlgReadlink()--Read Value of Symbolic Link for a description and an example of supplying the *path* in any CCSID.

buf

(Output) A pointer to the area in which the contents of the link should be stored.

This parameter will be returned 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.

bufsiz

(Input) The size of *buf* in bytes.

Authorities

Note: Adopted authority is not used.

Authorization required for readlink()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object	None	None

Return Value

value

readlink() was successful.

When *bufsiz* is greater than zero, the value returned is the number of bytes placed in the buffer. When *bufsiz* is zero, the value returned is the number of bytes contained in the symbolic link. The buffer is not changed.

If the return value is equal to *bufsiz*, the entire contents of the symbolic link may not have been returned. You can determine the size of the contents of the symbolic link by using either **lstat()** or **readlink()** with a zero value for *bufsiz*.

-1

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

Error Conditions

If **readlink()** 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.

The named file is not a symbolic link.

[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.

[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 do not support **readlink()**.

   - QSYS.LIB
   - Independent ASP QSYS.LIB
   - QDLS
   - QOPT
   - QNetWare
   - QNTC

Related Information

- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- lstat()--Get File or Link Information
- QlgReadlink()--Read Value of Symbolic Link
- stat()--Get File Information
- symlink()--Make Symbolic Link
- unlink()--Remove Link to File

Example

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

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

main() {
  char fn[]="readlink.file";
  char sl[]="readlink.symlink";
  char buf[30];
  int  file_descriptor;

  if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
```

```
  else {
    close(file_descriptor);
    if (symlink(fn, sl) != 0)
      perror("symlink() error");
    else {
      if (readlink(sl, buf, sizeof(buf)) < 0)
        perror("readlink() error");
      else printf("readlink() returned '%s' for '%s'\n", buf, sl);

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

Output:

```
readlink() returned 'readlink.file' for 'readlink.symlink'
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

readv()--Read from Descriptor Using Multiple Buffers

Syntax

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

int readv(int descriptor,
          struct iovec *io_vector[],
          int vector_length)
```

Threadsafe: Conditional; see Usage Notes.

The *readv()* function is used to receive data from a file or socket descriptor. *readv()* provides a way for data to be stored in several different buffers (*scatter/gather I/O*).

See read()--Read from Descriptor for more information related to reading from a descriptor.

Parameters

descriptor

(Input) The descriptor to be read. The descriptor refers to a file or a socket.

io_vector[]

(I/O) The pointer to an array of type **struct iovec**. **struct iovec** contains a sequence of pointers to buffers in which the data to be read is stored. The structure pointed to by the *io_vector* parameter is defined in **<sys/uio.h>**.

```
struct iovec {
   void      *iov_base;
   size_t   iov_len;
}
```

iov_base and *iov_len* are the only fields in *iovec* used by sockets. *iov_base* contains the pointer to a buffer and *iov_len* contains the buffer length. The rest of the fields are reserved.

vector_length

(Input) The number of entries in *io_vector*.

Authorities

No authorization is required.

Return Value

readv() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the number of bytes read.

Error Conditions

If **readv**() 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.

This may occur if *file_descriptor* refers to a socket and the socket is using a connection-oriented transport service, and a *connect()* was previously completed. The thread, however, does not have the appropriate privileges to the objects that were needed to establish a connection. For example, the *connect()* required the use of an APPC device that the thread was not authorized to.

[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 **readv** 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 *file_descriptor* refers to a socket that is using a connectionless transport service, is not a socket of type `SOCK_RAW`, and is not bound to an address.

The file resides in a file system that does not support large files, and the starting offset of the file exceeds 2 GB minus 2 bytes.

[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.

[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 is greater than or equal to 2GB minus 2 bytes.

[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.

[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.

When the descriptor refers to a socket, *errno* could indicate one of the following errors:

[ECONNABORTED] Connection ended abnormally.

This error code indicates that the transport provider ended the connection abnormally because of one of the following:

- The retransmission limit has been reached for data that was being sent on the socket.
- A protocol error was detected.

[ECONNREFUSED] The destination socket refused an attempted connect operation.

[ECONNRESET] A connection with a remote socket was reset by that socket.

[EINTR] Interrupted function call.

[ENOTCONN] Requested operation requires a connection.

This error code is returned only on sockets that use a connection-oriented transport service.

[ETIMEDOUT] A remote host did not respond within the timeout period.

A non-blocking **connect()** was previously completed that resulted in the connection timing out. No connection is established. This error code is returned only on sockets that use a connection-oriented transport service.

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

[EWOULDBLOCK] Operation would have caused the process to be suspended.

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

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. The *io_vector*[] parameter is an array of **struct iovec** structures. When a *readv()* is issued, the system processes the array elements one at a time, starting with *io_vector*[0]. For each element, **iov_len** bytes of received data are placed in storage pointed to by **iov_base**. Data is placed in storage until all buffers are full, or until there is no more data to receive. Only the storage pointed to by **iov_base** is updated. No change is made to the **iov_len** fields. To determine the end of the data, the application program must use the following:
   - The function return value (the total number of bytes received).
   - The lengths of the buffers pointed to by **iov_base**.
3. For sockets that use a connection-oriented transport service (for example, sockets with a type of SOCK_STREAM), a returned value of zero indicates one of the following:
   - The partner program has issued a *close()* for the socket.
   - The partner program has issued a *shutdown()* to disable writing to the socket.
   - The connection is broken and the error was returned on a previously issued socket function.
   - A *shutdown()* to disable reading was previously done on the socket.
4. The following applies to sockets that use a connectionless transport service (for example, a socket with a type of SOCK_DGRAM):
   - If a *connect()* has been issued previously, then data can be received only from the address specified in the previous *connect()*.
   - The address from which data is received is discarded, because the *readv()* has no address parameter.
   - The entire message must be read in a single read operation. If the size of the message is too large to fit in the user-supplied buffers, the remaining bytes of the message are discarded.
   - A returned value of zero indicates one of the following:
     - The partner program has sent a NULL message (a datagram with no user data).
     - A *shutdown()* to disable reading was previously done on the socket.

- The buffer length specified by the application was zero.

5. For the file systems that do not support large files, **readv()** 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, **readv()** will return [EOVERFLOW] if the starting offset exceeds 2GB minus 2 bytes and file was not opened for large file access.

6. 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.
7. QOPT File System Differences

   When reading from files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being read are ignored.
8. Using this function successfully on the /dev/null »or /dev/zero «character special file results in a return value of 0. In addition, the access time for the file is updated.

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
- read()--Read from Descriptor
- recv()--Receive Data
- recvfrom()--Receive Data
- recvmsg()--Receive Data or Descriptors or Both
- write()--Write to Descriptor
- writev()--Write to Descriptor Using Multiple Buffers

API introduced: V3R1

---

rename()--Rename File or Directory

Syntax

```
#include <Qp0lstdi.h>

int rename(const char *old, const char *new);
```

Threadsafe: Conditional; see Usage Notes.

The **rename()** function can be defined to be either **Qp0lRenameUnlink()** or **Qp0lRenameKeep()**, depending upon the definitions of the _POSIX_SOURCE and _POSIX1_SOURCE macros in the **<Qp0lstdi.h>** header file:

- When _POSIX_SOURCE or _POSIX1_SOURCE is defined, **rename()** is defined to be **Qp0lRenameUnlink()**. Either **rename()** or **Qp0lRenameUnlink()** can be used to rename a file or directory with the semantics of **Qp0lRenameUnlink()**.
- When _POSIX_SOURCE and _POSIX1_SOURCE are **not** defined, **rename()** is defined to be **Qp0lRenameKeep()**. Either **rename()** or **Qp0lRenameKeep()** can be used to rename a file or directory with the semantics of **Qp0lRenameKeep()**.

When the **<Qp0lstdi.h>** header file is **not** included, **rename()** operates only on database files in the QSYS.LIB »or independent ASP QSYS.LIB «file system, as it did before the introduction of the integrated file system.

For details on the use of **rename()**, see the **Qp0lRenameUnlink()** and **Qp0lRenameKeep()** functions.

Parameters

old

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

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.

new

(Input) A pointer to the null-terminated path name of the new name of the file.

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 file name is assumed to be represented in the language and country or region currently in effect for the process.

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 <**stdio.h**> file (see Header Files for UNIX-Type Functions)
- The <**Qp0lstdi.h**> file (see Header Files for UNIX-Type Functions)
- pathconf()--Get Configurable Path Name Variables
- Qp0lRenameKeep()--Rename File or Directory, Keep "new" If It Exists
- Qp0lRenameUnlink()--Rename File or Directory, Unlink "new" If It Exists

---

API introduced: V3R1

---

rewinddir()--Reset Directory Stream to Beginning

Syntax

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

void rewinddir(DIR *dirp);
```

Threadsafe: Yes

The **rewinddir()** function "rewinds" the position of an open directory stream to the beginning. *dirp* points to a DIR associated with an open directory stream.

The next call to **readdir()** reads the first entry in the directory. If the contents of the directory have changed since the directory was opened and **rewinddir()** is called, subsequent calls to **readdir()** read the changed contents.

Parameters

dirp

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

Authorities

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

Return Value

None.

Error Conditions

None.

Error Messages

The following messages may be sent from this function:

CPE3418 E

Possible APAR condition or hardware failure.

CPF1F05 E

Directory handle not valid.

CPF3CF2 E

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

Usage Notes

1. If the *dirp* argument passed to **rewinddir()** does not refer to an open directory, unexpected results could occur.
2. Files that are added to the directory after **opendir()** or **rewinddir()** may not be returned on calls to **readdir()**.

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
- closedir()--Close Directory

Example

The following example produces the contents of a directory by opening it, rewinding it, and closing it:

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

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

  if ((dir = opendir("/")) == NULL)
    perror("opendir() error");
  else {
    puts("contents of root:");
    while ((entry = readdir(dir)) != NULL)
```

```
      printf("%s ", entry->d_name);
    rewinddir(dir);
    puts("");
    while ((entry = readdir(dir)) != NULL)
      printf("%s ", entry->d_name);
    closedir(dir);
    puts("");
  }
}
```

Output:

```
contents of root:
  . .. QSYS.LIB QDLS QOpenSys QOPT home
  . .. QSYS.LIB QDLS QOpenSys QOPT home newdir
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

rmdir()--Remove Directory

Syntax

```
#include <unistd.h>

int rmdir(const char *path);
```

Threadsafe: Conditional; see Usage Notes.

The **rmdir()** function removes a directory, *path*, provided that the directory is empty; that is, the directory contains no entries other than "dot" (**.**) or "dot-dot" (**..**). *path* must not end in dot (**.**) or dot-dot (**..**).

If no job currently has the directory open, **rmdir()** deletes the directory itself. The space occupied by the directory is freed for new use. If one or more jobs have the directory open, **rmdir()** removes the link and the dot (**.**) or dot-dot (**..**). entries. The directory itself is not removed until the last job closes the directory. New files cannot be created under a directory after the last link is removed, even if the directory is still open.

rmdir() does not remove a directory that still contains files or subdirectories. If *path* refers to a directory that is not empty, the [ENOTEMPTY] error is returned. If *path* refers to the current directory of the current job, to the root (/) directory, or to a directory that cannot be removed, the [EBUSY] error is returned.

If *path* refers to a symbolic link, **rmdir()** does not affect any file or directory named by the contents of the symbolic link.

If **rmdir()** is successful, the change and modification times for the parent directory are updated.

Parameters

path

(Input) A pointer to the null-terminated path name of the directory to be removed.

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 QlgRmdir()--Remove Directory (using NLS-enabled path name) for a description and an example of supplying the *path* in any CCSID.

Authorities

Note: Adopted authority is not used.

Figure 1-70. Authorization Required for rmdir() (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 removed	*X	EACCES
Parent directory of the directory to be removed	*WX	EACCES
Directory to be removed	*OBJEXIST	EACCES

Figure 1-71. Authorization Required for rmdir() 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 removed	*X	EACCES
Parent directory of the directory to be removed	*X	EACCES
Directory to be removed, if it is a library	*OBJEXIST, *RX	EACCES
Directory to be removed, if it is a database file	*OBJEXIST, *OBJOPR	EACCES

Figure 1-72. Authorization Required for rmdir() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the directory to be removed	*X	EACCES
Parent directory of the directory to be removed	*X	EACCES
Directory to be removed	*OBJEXIST, *X	EACCES

Figure 1-73. Authorization Required for rmdir() in the QOPT File System

Object Referred to	Authority Required	errno
Volume authorization list	*CHANGE	EACCES
Each directory in the path name preceding the directory to be removed if volume media format is Universal Disk Format (UDF)	*X	EACCES
Parent directory of the directory to be removed if volume media format is Universal Disk Format (UDF)	*WX	EACCES
Directory to be removed if volume media format is Universal Disk Format (UDF)	*W	EACCES
Directory and parent directories if volume media format is not Universal Disk Format (UDF)	None	None

Return Value

0

rmdir() was successful.

-1

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

Error Conditions

If **rmdir**() 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 path cannot be removed because it is the current working directory of the current process, or it is currently being used by the system.

[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. The last component of *path* is 'dot' or 'dot-dot'.

[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. The last component of the path name is dot or dot-dot.

[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.

[ENOTEMPTY]

Directory not empty.

You tried to remove a directory that is not empty. A directory cannot contain objects when it is being removed.

The specified directory is not empty.

[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.

[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.

[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.

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

   If one or more jobs have the library or file open, **rmdir()** returns [EBUSY].

   If **rmdir()** is successful, the change and modification times for the parent library are updated only if the "directory" being removed is a database file.

3. QDLS File System Differences

If one or more jobs have the folder open, or are using the folder as their current directory, **rmdir()** returns [EBUSY].

4. QOPT File System Differences

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

   If *path* refers to a directory that any job has open, the [EBUSY] error is returned.

5. QNTC File System Differences

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

Related Information

- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- mkdir()--Make Directory
- QlgRmdir()--Remove Directory (using NLS-enabled path name)
- unlink()--Remove Link to File

Example

The following example removes a directory:

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

main() {
  char new_dir[]="new_dir";
  char new_file[]="new_dir/new_file";
  int  file_descriptor;

  if (mkdir(new_dir, S_IRWXU|S_IRGRP|S_IXGRP) != 0)
    perror("mkdir() error");
  else if ((file_descriptor = creat(new_file, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(file_descriptor);
    unlink(new_file);
  }

  if (rmdir(new_dir) != 0)
    perror("rmdir() error");
  else
    puts("removed!");
```

```
}
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

stat()--Get File Information

Syntax

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

int stat(const char *path, struct stat *buf);
```

Threadsafe: Conditional; see Usage Notes.

The **stat()** function gets status information about a specified file and places it in the area of memory pointed to by the *buf* argument.

If the named file is a symbolic link, **stat()** resolves the symbolic link. It also returns information about the resulting file.

Parameters

path

(Input) A pointer to the null-terminated path name of the file from which information is required.

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 QlgStat()--Get File Information (using NLS-enabled path name) 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.

The information is returned in the following stat structure, as defined in the <**sys/stat.h**> header file:

mode_t	st_mode	A bit string indicating the permissions and privileges of the file. Symbols are defined in the <sys/stat.h> header file to refer to bits in a mode_t value; these symbols are listed in chmod()--Change File Authorizations.
ino_t	st_ino	The file ID for the object. This number uniquely identifies the object within a file system. When st_ino and st_dev are used together, they uniquely identify the object on the system.
nlink_t	st_nlink	The number of links to the file. This field will be 65,535 if the value could not fit in the specified nlink_t field. The complete value will be in the st_nlink32 field.
unsigned short	st_reserved2	Reserved.

uid_t	st_uid	The numeric user ID (uid) of the owner of the file.
gid_t	st_gid	The numeric group ID (gid) for the file.
off_t	st_size	Defined as follows for each file type: *Regular File* The number of data bytes in the file. *Directory* The number of bytes allocated to the directory. *Symbolic Link* The number of bytes in the path name stored in the symbolic link. *Local Socket* Always zero. *OS/400 Native Object* This value is dependent on the object type.
time_t	st_atime	The most recent time the file was accessed.
time_t	st_mtime	The most recent time the contents of the file were changed.
time_t	st_ctime	The most recent time the status of the file was changed.
dev_t	st_dev	The file system ID to which the object belongs. This number uniquely identifies the file system to which the object belongs. When st_ino and st_dev are used together, they uniquely identify the object on the system. This field will be 4,294,967,295 if the value could not fit in the specified dev_t field. The complete value will be in the st_dev64 field.
size_t	st_blksize	The block size of the file in bytes.
unsigned long	st_allocsize	The number of bytes allocated to the file.
qp0l_objtype_t	st_objtype	The iSeries object type; for example, *STMF or *DIR. Refer to CL Programming for a list of the iSeries object types.
unsigned short	st_codepage	The code page derived from the CCSID used for the data in the file or the extended attributes of the directory. If the returned value of this field is zero (0), there is more than one code page associated with the st_ccsid. If the st_ccsid is not a supported iSeries CCSID, the st_codepage is set equal to the st_ccsid.
unsigned short	st_ccsid	The CCSID used for the data in the file or the extended attributes of the directory.

dev_t	st_rdev	The device ID of the object if the object is a character special file or block special file. This number uniquely identifies the file device. This field will be 4,294,967,295 if the value could not fit in the specified dev_t field. The complete value will be in the st_rdev64 field.
nlink32_t	st_nlink32	The number of links to the file.
dev64_t	st_rdev64	The device ID of the object in 64 bit format. See st_rdev for more information.
dev64_t	st_dev64	The file system ID to which the object belongs in 64 bit format. See st_dev for more information.
char	st_reserved1[36]	Reserved.
unsigned int	st_ino_gen_id	The generation ID associated with the file ID.

Values of time_t are given in terms of seconds since a fixed point in time called the *Epoch*.

You can examine properties of a mode_t value from the st_mode field using a collection of macros defined in the **<sys/stat.h>** header file. If *mode* is a mode_t value, then:

S_ISBLK(mode)

Is nonzero for block special files

S_ISCHR(mode)

Is nonzero for character special files

S_ISDIR(mode)

Is nonzero for directories

S_ISFIFO(mode)

Is nonzero for pipes and FIFO special files

S_ISREG(mode)

Is nonzero for regular files

S_ISLNK(mode) >

Is nonzero for symbolic links

S_ISSOCK(mode)

Is nonzero for local sockets

S_ISNATIVE(mode)

Is nonzero for OS/400 native objects

Authorities

Note: Adopted authority is not used.

Figure 1-74. Authorization Required for stat()>

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

Object	None	None

Return Value

0

stat() was successful. The information is returned in *buf*.

-1

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

Error Conditions

If **stat()** 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.

[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.

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

   The **stat()** function could return zero for the st_atime value (in the stat structure) under some conditions.

3. QDLS File System Differences

   If the date corresponding to the st_atime, st_mtime, or st_ctime value precedes 1970, **stat()** returns zero for that value. Also, if the specified *path* is /QDLS, **stat()** returns zero for all three values st_atime, st_mtime, and st_ctime.

4. 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).

   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 1-74, "Authorization Required for stat()." If the object exists on a volume formatted in some other media format, no authorization checks are made on the object or on each directory in the path name. The volume authorization list is checked for *USE authority regardless of the media format of the volume.

   stat on /QOPT will always return 2,147,483,647 for size fields.

   stat on optical volumes will return the volume capacity or 2,147,483,647, whichever is smaller.

   The file access time is not changed.

5. 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.)

6. 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.

7. This function will fail with the [EOVERFLOW] 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).
8. 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 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
- lstat()--Get File or Link Information
- mkdir()--Make Directory
- open()--Open File
- QlgStat()--Get File Information (using NLS-enabled path name)
- read()--Read from Descriptor
- readlink()--Read Value of Symbolic Link
- stat64()--Get File Information (Large File Enabled)
- symlink()--Make Symbolic Link
- 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:

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

```
#include <time.h>

main() {
  struct stat info;

  if (stat("/", &info) != 0)
    perror("stat() error");
  else {
    puts("stat() returned the following information about root f/s:");
    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);
  }
}
```

Output: note that the following information will vary from system to system.

```
stat() returned the following information about root f/s:
  inode:   0
 dev id:   1
   mode:   010001ed
  links:   3
    uid:   137
    gid:   500
```

API introduced: V3R1

---

stat64()--Get File Information (Large File Enabled)

Syntax

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

int stat64(const char *path, struct stat64 *buf);
```

Threadsafe: Conditional; see Usage Notes.

The **stat64()** function gets status information about a specified file and places it in the area of memory pointed to by the *buf* argument.

If the named file is a symbolic link, **stat64()** resolves the symbolic link. It also returns information about the resulting file.

stat64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 byte and returning correct sizes.

For additional information about authorities required, error conditions, and examples, see stat()--Get File Information.

Parameters

path

(Input) A pointer to the null-terminated path name of the file from which information is required.

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 QlgStat64()--Get File Information (large file enabled and using NLS-enabled path name) 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.

The information is returned in the following stat64 structure, as defined in the <**sys/stat.h**> header file:

mode_t	st_mode	A bit string indicating the permissions and privileges of the file. Symbols are defined in the <sys/stat.h> header file to refer to bits in a mode_t value; these symbols are listed in chmod()--Change File Authorizations.

ino_t	st_ino	The file ID for the object. This number uniquely identifies the object within a file system. When st_ino and st_dev are used together, they uniquely identify the object on the system.
uid_t	st_uid	The numeric user ID (uid) of the owner of the file.
gid_t	st_gid	The numeric group ID (gid) for the file.
off64_t	st_size	Defined as follows for each file type: *Regular File* The number of data bytes in the file. *Directory* The number of bytes allocated to the directory. *Symbolic Link* The number of bytes in the path name stored in the symbolic link. *Local Socket* Always zero. *OS/400 Native Object* This value is dependent on the object type.
time_t	st_atime	The most recent time the file was accessed.
time_t	st_mtime	The most recent time the contents of the file were changed.
time_t	st_ctime	The most recent time the status of the file was changed.
dev_t	st_dev	The file system ID to which the object belongs. This number uniquely identifies the file system to which the object belongs. When st_ino and st_dev are used together, they uniquely identify the object on the system. »This field will be 4,294,967,295 if the value could not fit in the specified dev_t field. The complete value will be in the st_dev64 field.«
size_t	st_blksize	The block size of the file in bytes.
nlink_t	st_nlink	The number of links to the file. »This field will be 65,535 if the value could not fit in the specified nlink_t field. The complete value will be in the st_nlink32 field.«
unsigned short	st_codepage	The code page derived from the CCSID used for the data in the file or the extended attributes of the directory. If the returned value of this field is 0, a code page could not be derived.
unsigned long long	st_allocsize	The number of bytes allocated to the file.
unsigned int	st_ino_gen_id	The generation ID associated with the file ID.

qp0l_objtype_t	st_objtype	The iSeries0 object type; for example, *STMF or *DIR. Refer to CL Programming for a list of the iSeries object types.
» char	st_reserved2[5]	Reserved.«
» dev-t	st_rdev	The device ID of the object if the object is a character special file or block special file. This number uniquely identifies the file device. This field will be 4,294,967,295 if the value could not fit in the specified dev_t field. The complete value will be in the st_rdev64 field. «
» dev64_t	st_rdev64	The device ID of the object in 64 bit format. See st_rdev for more information.«
» dev64_t	st_dev64	The file system ID to which the object belongs in 64 bit format. See st_dev for more information.«
» nlink32_t	st_nlink32	The number of links to the file.«
» char	st_reserved1[26]	Reserved.«
unsigned short	st_ccsid	The CCSID used for the data in the file or the extended attributes of the directory.

Values of time_t are given in terms of seconds since a fixed point in time called the *Epoch*.

You can examine properties of a mode_t value from the st_mode field using a collection of macros defined in the **<sys/stat.h>** header file. If *mode* is a mode_t value, then:

S_ISBLK(mode)

Is nonzero for block special files

S_ISCHR(mode)

Is nonzero for character special files

S_ISDIR(mode)

Is nonzero for directories

S_ISFIFO(mode)

Is nonzero for pipes and FIFO special files

S_ISREG(mode)

Is nonzero for regular files

S_ISLNK(mode)

Is nonzero for symbolic links

S_ISSOCK(mode)

Is nonzero for local sockets

S_ISNATIVE(mode)

Is nonzero for OS/400 native objects

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use either the **stat64()** API or the **QlgStat64()** API and the struct stat64 data type, you must compile the source with _LARGE_FILE_API defined.
2. All of the usage notes for **stat()** also apply to **stat64()** and to **QlgStat64()**. See Usage Notes in the **stat**() API.

---

statvfs()--Get File System Information

Syntax

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

int statvfs(const char *path, struct statvfs *buf);
```

Threadsafe: Conditional; see Usage Notes.

The **statvfs()** function gets status information about the file system that contains the file named by the *path* argument. The information will be placed in the area of memory pointed to by the *buf* argument.

If the named file is a symbolic link, **statvfs()** resolves the symbolic link.

Parameters

path

(Input) A pointer to the null-terminated path name of the file from which file system information is required.

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 QlgStatvfs()--Get File System Information (using NLS-enabled path name) 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.

The information is returned in the following statvfs structure, as defined in the <**sys/statvfs.h**> header file. Signed fields of the statvfs structure that are not supported by the mounted file system will be set to -1.

unsigned long	f_bsize	The file system block size in bytes. Some file systems may return zero in this field. If this field is zero, then the contents of the f_blocks, f_bfree, and f_bavail fields are undefined.
unsigned long	f_frsize	The fundamental file system block size in bytes. Some file systems may return zero in this field. If this field is zero, then the contents of the f_blocks, f_bfree, and f_bavail fields are undefined.
_Bin8	f_blocks	The total number of blocks in the file system in terms of f_frsize.
_Bin8	f_bfree	The total number of free blocks in the file system.

_Bin8	f_bavail	The total number of free blocks available to a non-privileged process.
unsigned long	f_files	The total number of file serial numbers.
unsigned long	f_ffree	The total number of free file serial numbers.
unsigned long	f_favail	The number of free file serial numbers available to a non-privileged process.
unsigned long	f_fsid	The file system ID. This field will be 4,294,967,295 if the value could not fit in the specified unsigned long field.
unsigned long	f_flag	File system flags. Symbols are defined in the <**sys/statvfs.h**> header file to refer to bits in this field (see The f_flags field).
unsigned long	f_namemax	The maximum file name length in the file system. Some file systems may return the maximum value that can be stored in an unsigned long to indicate the file system has no maximum file name length. The maximum value that can be stored in an unsigned long is defined in <**limits.h**> as ULONG_MAX. 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.
unsigned long	f_pathmax	The maximum path length in the file system. Some file systems may return the maximum value that can be stored in an unsigned long to indicate the file system has no maximum path length. The maximum value that can be stored in an unsigned long is defined in <**limits.h**> as ULONG_MAX. 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.
long	f_objlinkmax	The maximum number of hard links for objects other than directories.
long	f_dirlinkmax	The maximum number of hard links for a directory.
char	f_reserved1[4]	Reserved.
unsigned long long	f_fsid64	The file system ID in 64 bit format.
char	f_basetype[80]	The NULL-terminated file system type name. The text in this field will be returned in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this is assumed to be represented in the default CCSID of the job.

The f_flags field

The following symbols are defined in the <**sys/statvfs.h**> header file to refer to bits that may be returned in the f_flags field:

ST_RDONLY

The file system is mounted for read-only access.

ST_NOSUID

The file system does not support setuid/setgid semantics.

ST_CASE_SENSITIVE

The file system is case sensitive.

ST_CHOWN_RESTRICTED

The file system restricts the changing of the owner or primary group to a process that has the appropriate privileges.

ST_THREAD_SAFE

The file system is thread-safe. Thread-safe APIs may operate on objects in this file system in a thread-safe manner.

ST_DYNAMIC_MOUNT

The file system allows itself to be dynamically mounted and unmounted.

ST_NO_MOUNT_OVER

The file system does not allow any part of it to be mounted over.

ST_NO_EXPORTS

The file system does not allow any of its objects to be exported to the Network File System (NFS) Server.

ST_SYNCHRONOUS

The file system supports the "synchronous write" semantic of NFS Version 2.

Authorities

Note: Adopted authority is not used.

Figure 1-75. Authorization Required for statvfs()

Object Referred to	Authority Required	errno
Each directory in the path name that precedes the object	*X	EACCES
Object	None	None

Return Value

0

statvfs() was successful. The information is returned in *buf*.

-1

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

Error Conditions

If **statvfs()** 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.

[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. 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 this function is compiled with _LARGE_FILES defined, it will be mapped to **statvfs64()**. 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
- fstatvfs()--Get File System Information by Descriptor
- link()--Create Link to File
- open()--Open File
- QlgStatvfs()--Get File System Information (using NLS-enabled path name)
- read()--Read from Descriptor
- statvfs64()--Get File System Information (64-Bit Enabled)
- 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;

  if (-1 == statvfs("/", &info))
    perror("statvfs() error");
  else {
    puts("statvfs() 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

---

statvfs64()--Get File System Information (64-Bit Enabled)

Syntax

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

int statvfs64(const char *path, struct statvfs64 *buf)
```

Threadsafe: Conditional; see Usage Notes.

The **statvfs64()** function gets status information about the file system that contains the file named by the *path* argument. The information is placed in the area of memory pointed to by the *buf* argument.

If the named file is a symbolic link, **statvfs64()** resolves the symbolic link.

For details about authorities required, error conditions, and examples, see statvfs()--Get File System Information.

Parameters

path

(Input) A pointer to the null-terminated path name of the file from which file system information is required.

This parameter is assumed to be represented in the coded character set identifier (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 QlgStatvfs64()--Get File System Information (64-Bit enabled and using NLS-enabled path name) 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.

The information is returned in the following statvfs64 structure, as defined in the <**sys/statvfs.h**> header file. Signed fields of the statvfs64 structure that are not supported by the mounted file system will be set to -1.

unsigned long	f_bsize	The file system block size in bytes. Some file systems may return zero in this field. If this field is zero, then the contents of the f_blocks, f_bfree, and f_bavail fields are undefined.
unsigned long	f_frsize	The fundamental file system block size in bytes. Some file systems may return zero in this field. If this field is zero, then the contents of the f_blocks, f_bfree, and f_bavail fields are undefined.

unsigned long long	f_blocks	The total number of blocks in the file system in terms of f_frsize.
unsigned long long	f_bfree	The total number of free blocks in the file system.
unsigned long long	f_bavail	The total number of free blocks available to a non-privileged process.
unsigned long	f_files	The total number of file serial numbers.
unsigned long	f_ffree	The total number of free file serial numbers.
unsigned long	f_favail	The number of free file serial numbers available to a non-privileged process.
unsigned long	f_fsid	The file system ID. This field will be 4,294,967,295 if the value could not fit in the specified unsigned long field.
unsigned long	f_flag	File system flags. Symbols are defined in the <**sys/statvfs.h**> header file to refer to bits in this field (see The f_flags field).
unsigned long	f_namemax	The maximum file name length in the file system. Some file systems may return the maximum value that can be stored in an unsigned long to indicate the file system has no maximum file name length. The maximum value that can be stored in an unsigned long is defined in <**limits.h**> as ULONG_MAX. 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.
unsigned long	f_pathmax	The maximum path length in the file system. Some file systems may return the maximum value that can be stored in an unsigned long to indicate the file system has no maximum path length. The maximum value that can be stored in an unsigned long is defined in <**limits.h**> as ULONG_MAX. 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.
long	f_objlinkmax	The maximum number of hard links for objects other than directories.
long	f_dirlinkmax	The maximum number of hard links for a directory.
char	f_reserved1[4]	Reserved.
unsigned long long	f_fsid64	The file system ID in 64 bit format.

char	f_basetype[80]	The NULL-terminated file system type name. The text in this field will be returned in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this is assumed to be represented in the default CCSID of the job.

For further details about the f_flags field, see statvfs()--Get File System Information.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **statvfs64()** API or the **QlgStatvfs64()** API and the struct statvfs64 data type, you must compile the source with the _LARGE_FILE_API macro defined.
2. All of the usage notes for **statvfs()** apply to **statvfs64()** and **QlgStatvfs64()**. See Usage Notes in the **statvfs()** API.

---

symlink()--Make Symbolic Link

Syntax

```
#include <unistd.h>

int symlink(const char *pname, const char *slink);
```

Threadsafe: Conditional; see Usage Notes.

The **symlink()** function creates the symbolic link named by *slink* with the value specified by *pname*. File access checking is not performed on the file *pname*, and the file need not exist. In addition, a symbolic link can cross file system boundaries.

If *slink* names a symbolic link, **symlink()** fails with the [EEXIST] error.

A symbolic link path name is resolved in the following manner:

- When a component of a path name refers to a symbolic link rather than to a directory, the path name contained in the symbolic link is resolved.
- If the path name in the symbolic link begins with / (slash), the symbolic link path name is resolved relative to the root directory for the job.

  If the path name in the symbolic link does not start with / (slash), the symbolic link path name is resolved relative to the directory that contains the symbolic link.

- If the symbolic link is the last component of a path name, it may or may not be resolved. Resolution depends on the function using the path name. For example, **rename()** does not resolve a symbolic link when the symbolic link is the final component of either the new or old path name. However, **open()** does resolve a symbolic link when the link is the last component.
- If the symbolic link is not the last component of the original path name, remaining components of the original path name are resolved relative to the symbolic link.
- When a / (slash) is the last component of a path name and it is preceded by a symbolic link, the symbolic link is always resolved.

Any files and directories to which a symbolic link refers are checked for access permission.

symlink() sets the access, change, modification, and creation times for the new symbolic link. It also sets the change and modification times for the directory that contains the new symbolic link.

Parameters

pname

(Input) A pointer to the null-terminated value of the symbolic link.

The value of the symbolic link 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 QlgSymlink--Make Symbolic Link (using NLS-enabled path name) for a description and an example of supplying the *pname* in any CCSID.

slink

(Input) A pointer to the null-terminated name of the symbolic link to be created.

This parameter is assumed to be represented in the CCSID, language, and country or region 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 QlgSymlink--Make Symbolic Link (using NLS-enabled path name) for a description and an example of supplying the *slink* in any CCSID.

Authorities

Note: Adopted authority is not used.

Figure 1-76. Authorization Required for symlink()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be created	*X	EACCES
Parent directory of object to be created	*WX	EACCES

Return Value

0

symlink() was successful.

-1

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

Error Conditions

If **symlink()** 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.

[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.

[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.

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 do not support **symlink()**:
   - QSYS.LIB
   - Independent ASP QSYS.LIB
   - QDLS
   - QOPT
   - QFileSvr.400
   - QNetWare
   - QNTC

Related Information

- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- link()--Create Link to File
- QlgSymlink--Make Symbolic Link (using NLS-enabled path name)
- readlink()--Read Value of Symbolic Link
- unlink()--Remove Link to File

Example

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

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

main() {
  char fn[]="readlink.file";
  char sl[]="readlink.symlink";
  char buf[30];
  int  file_descriptor;
```

```
  if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(file_descriptor);
    if (symlink(fn, sl) != 0)
      perror("symlink() error");
    else {
      if (readlink(sl, buf, sizeof(buf)) < 0)
        perror("readlink() error");
      else printf("readlink() returned '%s'
      for '%s'\n", buf, sl);

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

Output:

```
readlink() returned 'readlink.file' for 'readlink.symlink'
```

API introduced: V3R1

---

sysconf()--Get System Configuration Variables

Syntax

```
#include <unistd.h>

long sysconf(int name);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Yes

The **sysconf()** function returns the value of a system configuration option. The configuration option to be obtained is specified by **name**.

Parameters

name

(Input) The named variable whose value is to be returned.

The value of **name** can be any one of the following symbols defined in the <**unistd.h**> header file, each corresponding to a system configuration option:

_SC_ARG_MAX	(Not supported by the iSeries server). Represents ARG_MAX, which indicates the maximum number of bytes of arguments and environment data that can be passed in an **exec** function.
_SC_CHILD_MAX	(Not supported by the iSeries server). Represents CHILD_MAX, which indicates the maximum number of jobs that a real user ID (uid) can have running simultaneously.
_SC_CLK_TCK	Represents the CLK_TCK macro, which indicates the number of clock ticks in a second. CLK_TCK is defined in the <**time.h**> header file.
_SC_JOB_CONTROL	(Not supported by the iSeries server). Represents the _POSIX_JOB_CONTROL macro, which indicates that certain job control operations are implemented by this version of the operating system. If _POSIX_JOB_CONTROL is defined (in the <**unistd.h**> header file), various APIs, such as **setpgid()**, provide more function than when the macro is not defined.
_SC_NGROUPS_MAX	Represents NGROUPS_MAX, which indicates the maximum number of supplementary group IDs (gid) that can be associated with a job.

_SC_OPEN_MAX	Represents OPEN_MAX, which indicates the maximum number of files that a single job can have open at one time.
_SC_PAGESIZE	Represents the system hardware page size. The symbol _SC_PAGESIZE is defined as the decimal value 11.
_SC_PAGE_SIZE	Represents the system hardware page size. The symbol _SC_PAGE_SIZE is defined as the decimal value 12.
_SC_SAVED_IDS	(Not supported by the iSeries server). Represents the _POSIX_SAVED_IDS macro, which indicates that this POSIX implementation has a saved set uid and a saved set gid. If the macro exists, it is defined in the <**unistd.h**> header file. This symbol affects the behavior of such functions as **setuid()** and **setgid()**.
_SC_STREAM_MAX	Represents the STREAM_MAX macro, which indicates the maximum number of streams that a job can have open at one time. The macro is defined in the <**limits.h**> header file.
_SC_TZNAME_MAX	(Not supported by the iSeries server). Represents the TZNAME_MAX macro, which indicates the maximum length of the name of a time zone. If the macro exists, it is defined in the <**limits.h**> header file.
_SC_VERSION	(Not supported by the iSeries server). Represents the _POSIX_VERSION macro, which indicates the version of the POSIX.1 standard that the system conforms to. If the macro exists, it is defined in the <**unistd.h**> header file.
_SC_CCSID	Represents the default coded character set identifier (CCSID) used internally for integrated file system path names. A CCSID uniquely identifies the coded graphic character representation of a path name and includes such information as the character set and code page identifier. The symbol _SC_CCSID is defined as the decimal value 10.

Authorities

No authorization is required.

Return Value

value	**sysconf()** was successful. The value associated with the specified option is returned.

-1 One of the following has occurred:

- The variable corresponding to **name** is valid but is not supported by the system. The **errno** global variable is not changed.
- **sysconf()** failed in some other way. The **errno** is set to indicate the error.

Error Conditions

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

[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 value for *name* is not valid.

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.

Related Information

- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)

Example

See Code disclaimer information for information pertaining to code examples.

The following example determines the value of OPEN_MAX:

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

main() {
  long result;

  errno = 0;
  puts("examining OPEN_MAX limit");
  if ((result = sysconf(_SC_OPEN_MAX)) == -1)
    if (errno == 0)
      puts("OPEN_MAX is not supported.");
    else perror("sysconf() error");
  else
    printf("OPEN_MAX is %ld\n", result);
}
```

Output:

```
examining OPEN_MAX limit
OPEN_MAX is 200
```

API introduced: V3R1

---

umask()--Set Authorization Mask for Job

Syntax

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

mode_t umask(mode_t cmask);
```

Threadsafe: Yes

Every job has a file creation mask. When a job starts, the value of the file creation mask is zero. The value of zero means that no permissions are masked when a file or directory is created in the job. The **umask()** function changes the value of the file creation mask for the current job to the value specified in *cmask*.

The *cmask* argument controls file permission bits that should be set whenever the job creates a file. File permission bits set to 1 in the file creation mask are set to 0 in the file permission bits of files that are created by the job.

For example, if a call to **open()** specifies a *mode* argument with file permission bits, the file creation mask of the job affects the *mode* argument; bits that are 1 in the mask are set to 0 in the *mode* argument and, therefore, in the mode of the created file.

Only the file permission bits of *cmask* are used. The other bits in *cmask* must be cleared (not set), or the CPFA0D3 message is issued.

Parameters

cmask

(Input) The new value of the file creation mask. For a description of the permission bits, see chmod()--Change File Authorizations.

Authorities

No authorization is required.

Return Value

umask() returns the previous value of the file creation mask. It does not return -1 or set the *errno* global variable.

Error Conditions

None.

Error Messages

The following messages may be sent from this function:

CPE3418 E

Possible APAR condition or hardware failure.

CPFA0D3 E

cmask parameter is not valid.

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. QNTC File System Differences

   umask() does not update the file creation mask for QNTC. The settings specified in *cmask* are ignored.

Related Information

- The <**sys/stat.h**> file (see Header Files for UNIX-Type Functions)
- chmod()--Change File Authorizations
- creat()--Create or Rewrite File
- mkdir()--Make Directory
- open()--Open File

Example

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

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

```
main()
{
  int  file_descriptor;
  struct stat info;

  umask(S_IRWXG);

  if ((file_descriptor =
       creat("umask.file", S_IRWXU|S_IRWXG)) < 0)
     perror("creat() error");
  else {
    fstat(file_descriptor, &info);
    printf("permissions are: %08x\n", info.st_mode);
    close(file_descriptor);
    unlink("umask.file");
  }
}
```

Output:

```
permissions are: 000081c0
```

API introduced: V3R1

---

unlink()--Remove Link to File

Syntax

```
#include <unistd.h>

int unlink(const char *path);
```

Threadsafe: Conditional; see Usage Notes.

The **unlink()** function removes a directory entry that refers to a file. This **unlink()** deletes the link named by *path* and decrements the link count for the file itself.

If the link count becomes zero and no job currently has the file open, the file itself is deleted. The space occupied by the file is freed for new use, and the current contents of the file are lost. If one or more jobs have the file open when the last link is removed, **unlink()** removes the link, but the file itself is not removed until the last job closes the file.

unlink() cannot be used to remove a directory; use **rmdir()** instead.

If *path* refers to a symbolic link, **unlink()** removes the symbolic link but not a file or directory named by the contents of the symbolic link.

If **unlink()** succeeds, the change and modification times for the parent directory are updated. If the link count of the file is not zero, the change time for the file is also updated. If **unlink()** fails, the link is not removed.

If the file is checked out, **unlink()** fails with the [EBUSY] error. If the file is marked "read-only", **unlink()** fails with the [EROOBJ] error.

Parameters

path

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

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 QlgUnlink()--Remove Link to File (using NLS-enabled path name) for a description and an example of supplying the *path* in any CCSID.

Authorities

Note: Adopted authority is not used.

Figure 1-77. Authorization Required for unlink() (excluding QSYS.LIB, independent ASP QSYS.LIB, QDLS and QOPT)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be unlinked	*X	EACCES
Parent directory of the object to be unlinked	*WX	EACCES
Object to be unlinked	*OBJEXIST	EACCES

Figure 1-78. Authorization Required for unlink() 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 unlinked	*X	EACCES
Parent directory of the object to be unlinked	See **Note**	EACCES
Object to be unlinked	See **Note**	EACCES
Note: The required authorization varies for each object type. See the DLTxxx commands in the iSeries Security Reference book for details.		

Figure 1-79. Authorization Required for unlink() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be unlinked	*X	EACCES
Parent directory of the object to be unlinked	*X	EACCES
Object to be unlinked	*ALL	EACCES

Figure 1-80. Authorization Required for unlink() 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 to be unlinked if volume media format is Universal Disk Format (UDF)	*X	EACCES
Parent directory of the object to be unlinked if volume media format is Universal Disk Format (UDF)	*WX	EACCES
Object to be unlinked if volume media format is Universal Disk Format (UDF)	*W	EACCES
Object to be unlinked and parent directories if volume media format is not Universal Disk Format (UDF)	None	None

Return Value

0

unlink() was successful.

-1

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

Error Conditions

If **unlink()** 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 file may be checked out.

[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.

[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.

[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.

[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.

[EPERM]

Operation not permitted.

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

requested operation.

unlink() is not permitted on directories in this part of the directory hierarchy, or **unlink()** is permitted but the user does not have sufficient authority.

[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.

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

The link to a file cannot be removed when a job has the file open.

The following object types cannot be unlinked when there are secondary threads active in the job: *CFGL, *CNNL, *COSD, *CTLD, *DEVD, *IPXD, *LIND, *MODD, *NTBD, *NWID, *NWSD. The operation will fail with error code [ENOTSAFE].

3. QDLS File System Differences

   The link to a document cannot be removed when a job has the document open (returns the [EBUSY] error).

4. QOPT File System Differences

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

   The link to a file cannot be removed when a job has the file open.

Related Information

- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- close()--Close File or Socket Descriptor
- link()--Create Link to File
- open()--Open File
- QlgOpen()--Open a File (using NLS-enabled path name)
- QlgRmdir()--Remove Directory (using NLS-enabled path name)
- QlgUnlink()--Remove Link to File (using NLS-enabled path name)
- rmdir()--Remove Directory

Example

The following example removes a link to a file:

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

main() {
  int file_descriptor;
  char fn[]="unlink.file";

  if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(file_descriptor);
```

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

API introduced: V3R1

---

utime()--Set File Access and Modification Times

Syntax

```
#include <utime.h>

int utime(const char *path, const struct utimbuf *times);
```

Threadsafe: Conditional; see Usage Notes.

The **utime()** function sets the access and modification times of *path* to the values in the utimbuf structure. If *times* is a NULL pointer, the access and modification times are set to the current time. If the named file is a symbolic link, **utime()** resolves the symbolic link.

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

When **utime()** completes successfully, it marks the change time of the file to be updated.

Parameters

path

(Input) A pointer to the null-terminated path name of the file for which the times should be 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 QlgUtime()--Set File Access and Modification Times (using NLS-enabled path name) for a description and an example of supplying the *path* in any CCSID.

times

(Input) A pointer to a structure utimbuf, which contains the times to be updated.

The structure utimbuf is defined according to the POSIX.1 definition as follows:

```
struct utimbuf {
   time_t   actime;   /* The new access time       */
   time_t   modtime;  /* The new modification time */
   }
```

The time_t type gives the number of seconds since the *Epoch*.

Authorities

Note: Adopted authority is not used.

Figure 1-81. Authorization Required for utime() (excluding QDLS)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object when changing the time to a specified value	Owner or *W (See **Note**)	EPERM
Object when changing the time to the current time	Owner or *W (See **Note**)	EACCES
Note: You do not need the listed authority if you have *ALLOBJ special authority.		

Figure 1-82. Authorization Required for utime() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object when changing the time to a specified value	*W	EPERM
Object when changing the time to the current time	*W	EACCES

Return Value

0

utime() was successful. The file access and modification times are changed.

-1

utime() was not successful. The file times are not changed. The *errno* global variable is set to indicate the error.

Error Conditions

If **utime()** 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. *times* is NULL and the job does not have authority to perform the requested function.

[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.

[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.

[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.

[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.

times is not NULL and 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 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

   These file systems do not support **utime**().

3. QDLS File System Differences

   Changing the times of the /QDLS directory (the root folder) is not allowed.

4. QOPT File System Differences

   The QOPT file system does not support **utime**().

5. QNTC File System Differences

   The QNTC file system does not set the access and modification times of *path*. The values in the utimbuf structure are ignored.

Related Information

- The <**utime.h**> file (see Header Files for UNIX-Type Functions)
- The <**limits.h**> file (see Header Files for UNIX-Type Functions)
- QlgUtime()--Set File Access and Modification Times (using NLS-enabled path name)

Example

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

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

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

  if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(file_descriptor);
    puts("before utime()");
    stat(fn,&info);
    printf("  utime.file modification time is %ld\n",
           info.st_mtime);
    ubuf.modtime = 0;       /* set modification time to Epoch */
    time(&ubuf.actime);
    if (utime(fn, &ubuf) != 0)
      perror("utime() error");
    else {
      puts("after utime()");
      stat(fn,&info);
      printf("  utime.file modification time is %ld\n",
             info.st_mtime);
    }
    unlink(fn);
  }
}
```

Output:

```
before utime()
  utime.file modification time is 749323571
after utime()
  utime.file modification time is 0
```

API introduced: V3R1

---

write()--Write to Descriptor

Syntax

```
#include <unistd.h>

ssize_t write
  (int file_descriptor, const void *buf, size_t nbyte);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **write()** function writes *nbyte* bytes from *buf* to the file or socket associated with *file_descriptor*. *nbyte* should not be greater than INT_MAX (defined in the <**limits.h**> header file). If *nbyte* is zero, **write()** simply returns a value of zero without attempting any other action.

If *file_descriptor* refers to a "regular file" (a stream file that can support positioning the file offset) or any other type of file on which the job can do an **lseek()** operation, **write()** begins writing at the file offset associated with *file_descriptor*, unless O_APPEND is set for the file (see below). A successful **write()** increments the file offset by the number of bytes written. If the incremented file offset is greater than the previous length of the file, the length of the file is set to the new file offset.

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

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

If **write()** is successful and *nbyte* is greater than zero, the change and modification times for the file are updated.

If *file_descriptor* refers to a descriptor obtained using the **open()** function with O_TEXTDATA specified, the data is written to the file assuming it is in textual form. The maximum number of bytes on a single write that can be supported for text data is 2,147,483,408 (2GB - 240) bytes. The data is converted from the code page of the application, job, or system to the code page of the file as follows:

- When writing to a true stream file, any line-formatting characters (such as carriage return, tab, and end-of-file) are just converted from one code page to another.
- When writing to a record file that is being used as a stream 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.

There are some important considerations if O_CCSID was specified on the **open()**.

- The **write()** will attempt to convert all of the data in the user's buffer. Successfully converted data will be written. Unconverted data is usually assumed to be a partial character. Partial characters will be buffered internally and data from the next consecutive write will be appended to the buffered data. If incorrect data is provided on a consecutive write, the write may fail with the [ECONVERT] error.

If an **lseek()** is performed, the file is closed, or the current job is ended, the buffered data will be discarded. Discarded data will not be written to the file. See lseek()--Set File Read/Write Offset for more information.

- Because of the above consideration and because of the possible expansion or contraction of converted data, applications using the O_CCSID flag should avoid assumptions about data size and the current file offset. For example, the user may supply a buffer to 100 bytes, but after an application has written the buffer to a new file, the file size may be 50, 200, or something else, depending on the CCSIDs involved.

If O_TEXTDATA was not specified on the **open()**, the data is written to the file without conversion. The application is responsible for handling the data.

When *file_descriptor* refers to a socket, the **write()** function writes to the socket identified by the socket descriptor.

Note: When the write completes successfully, the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode will be cleared. If the write is unsuccessful, the bits are undefined.>

Write requests to a pipe or FIFO are handled the same as a regular file, with the following exceptions:

- The S_ISUID and S_ISGID file mode bits will not be cleared.
- There is no file offset associated with a pipe or FIFO. Each write request will append to the end of the pipe or FIFO.
- Write requests of [PIPE_BUF] bytes or less will not be interleaved with data from other threads performing writes on the same pipe or FIFO. Writes of greater than [PIPE_BUF] bytes may have data interleaved on arbitrary boundaries with writes by other threads, whether or not the O_NONBLOCK flag of the file status flags is set.
- If the O_NONBLOCK flag was not specified and the pipe or FIFO is full, the write request will block the calling thread until the requested amount of data in *nbyte* is written.
- If the O_NONBLOCK flag was specified, then the following pertain to various write requests:
    - The **write()** function will not block the calling thread.
    - A write request for [PIPE_BUF] or fewer bytes will have the following effect:

      If there is sufficient space available in the pipe or FIFO, **write()** will transfer all the data and return the number of bytes requested. If there is not sufficient space in the pipe or FIFO, **write()** will transfer no data, return -1, and set *errno* to [EAGAIN].
    - A write request for more than [PIPE_BUF] bytes will cause one of the following:
        - When at least one byte can be written, **write()** will transfer what it can and return the number of bytes written.
        - When no data can be written, **write()** will transfer no data, return -1, and set *errno* to [EAGAIN].

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.

Authorities

No authorization is required.

Return Value

value	**write()** was successful. The value returned is the number of bytes actually written. This number is less than or equal to *nbyte*.
-1	**write()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **write()** 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.

If writing to a socket, this error code indicates one of the following:

- The destination address specified is a broadcast address and the socket option SO_BROADCAST was not set (with a *setsockopt()*).

- The process does not have the appropriate privileges to the destination address. This error code can only be returned on a socket with an address family of AF_INET and a type of SOCK_DGRAM.

[EAGAIN]
: Operation would have caused the process to be suspended.

If *file_descriptor* refers to a pipe or FIFO that has its O_NONBLOCK flag set, this error occurs if the **write()** would have blocked the calling thread.

[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 **write()** 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.

[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.

[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.

[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, thenretry the operation.

When the descriptor refers to a socket, *errno* could indicate one of the following errors:

[ECONNREFUSED] The destination socket refused an attempted connect operation.

This error code can only be returned on sockets that use a connectionless transport service.

[EDESTADDRREQ] Operation requires destination address.

A destination address has not been associated with the socket pointed to by the *fildes* parameter. This error code can only be returned on sockets that use a connectionless transport service.

[EHOSTDOWN] A remote host is not available.

This error code can only be returned on sockets that use a connectionless transport service.

[EHOSTUNREACH]	A route to the remote host is not available. This error code can only be returned on sockets that use a connectionless transport service.
[EINTR]	Interrupted function call.
[EMSGSIZE]	Message size out of range. The data to be sent could not be sent atomically because the size specified by *nbyte* is too large.
[ENETDOWN]	The network is not currently available. This error code can only be returned on sockets that use a connectionless transport service.
[ENETUNREACH]	Cannot reach the destination network. This error code can only be returned on sockets that use a connectionless transport service.
[ENOBUFS]	There is not enough buffer space for the requested operation.
[ENOTCONN]	Requested operation requires a connection. This error code can only be returned on sockets that use a connection-oriented transport service.
[EPIPE]	Broken pipe.
[EUNATCH]	The protocol required to support the specified address family is not available at this time.
[EWOULDBLOCK]	Operation would have caused the thread to be suspended.

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. 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 **write()** 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 **write()**'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 **write()** 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. 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. Sockets Usage Notes

   1. **write()** only works with sockets on which a **connect()** has been issued, since it does not allow the caller to specify a destination address.

   2. To broadcast on an AF_INET socket, the socket option SO_BROADCAST must be set (with a **setsockopt**()).

   3. When using a connection-oriented transport service, all errors except [EUNATCH] and [EUNKNOWN] are mapped to [EPIPE] on an output operation when either of the following occurs:

      - A connection that is in progress is unsuccessful.
      - An established connection is broken.

      To get the actual error, use **getsockopt()** with the SO_ERROR option, or perform an input operation (for example, **read**()).

7. For the file systems that do not support large files, **write()** 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, **write**() will return [EFBIG] if the starting offset exceeds 2GB minus 2 bytes and the file was not opened for large file access.

8. 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 character special file. In addition, the change and modification times for the file are updated.

9. » If the write exceeds the process soft file size limit, signal SIFXFSZ is issued. «

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)
- 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
- send()--Send Data
- sendmsg()--Send Data or Descriptors or Both
- sendto()--Send Data
- writev()--Write to Descriptor Using Multiple Buffers

Example

See Code disclaimer information for information pertaining to code examples.

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;
  char fn[]="write.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, '0', 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 (close(file_descriptor)!= 0)
       perror("close() error");
    if (unlink(fn)!= 0)
       perror("unlink() error");
  }
}
```

Output:

```
write() wrote 1000000 bytes
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

writev()--Write to Descriptor Using Multiple Buffers

Syntax

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

int writev(int descriptor,
           struct iovec *io_vector[],
           int vector_length)
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The *writev()* function is used to write data to a file or socket descriptor. *writev()* provides a way for the data that is going to be written to be stored in several different buffers (*scatter/gather I/O*).

Note: When the write completes successfully, the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode will be cleared. If the write is unsuccessful, the bits are undefined.

See write()--Write to Descriptor for more information related to writing to a descriptor.

Parameters

descriptor

(Input) The descriptor to which the data is to be written. The descriptor refers to either a file or a socket.

io_vector[]

(Input) The pointer to an array of type **struct iovec**. **struct iovec** contains a sequence of pointers to buffers in which the data to be written is stored. The structure pointed to by the *io_vector* parameter is defined in **<sys/uio.h>**.

```
struct iovec {
   void      *iov_base;
   size_t   iov_len;
}
```

iov_base and *iov_len* are the only fields in *iovec* used by sockets. *iov_base* contains the pointer to a buffer and *iov_len* contains the buffer length. The rest of the fields are reserved.

vector_length

(Input) The number of entries in *io_vector*.

Authorities

No authorization is required.

Return Value

writev() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the number of bytes written.

Error Conditions

If **writev()** 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.

If writing to a socket, this error code indicates one of the following:

- The destination address specified is a broadcast address and the socket option SO_BROADCAST was not set (with a *setsockopt()*).
- The process does not have the appropriate privileges to the destination address. This error code can only be returned on a socket with an address family of AF_INET and a type of SOCK_DGRAM.

[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 **writev()** 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 2GB 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 resides in a file system that does not support large files, and the starting offset exceeds 2GB minus 2 bytes.

[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.

[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.

»*[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.

[ETRUNC] Data was truncated on an input, output, or update operation.

[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.

When the descriptor refers to a socket, *errno* could indicate one of the following errors:

[ECONNREFUSED] The destination socket refused an attempted connect operation.

This error code can only be returned on sockets that use a connectionless transport service.

[EDESTADDRREQ] Operation requires destination address.

A destination address has not been associated with the socket pointed to by the *fildes* parameter. This error code can only be returned on sockets that use a connectionless transport service.

[EHOSTDOWN] A remote host is not available.

This error code can only be returned on sockets that use a connectionless transport service.

[EHOSTUNREACH] A route to the remote host is not available.

This error code can only be returned on sockets that use a connectionless transport service.

[EINTR] Interrupted function call.

[EMSGSIZE] Message size out of range.

The data to be sent could not be sent atomically because the size specified by *nbyte* is too large.

[ENETDOWN] The network is not currently available.

This error code can only be returned on sockets that use a connectionless transport service.

[ENETUNREACH] Cannot reach the destination network.

This error code can only be returned on sockets that use a connectionless transport service.

[ENOBUFS] There is not enough buffer space for the requested operation.

[ENOTCONN] Requested operation requires a connection.

This error code can only be returned on sockets that use a connection-oriented transport service.

[EPIPE] Broken pipe.

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

[EWOULDBLOCK] Operation would have caused the thread to be suspended.

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

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. *writev()* only works with sockets on which a *connect()* has been issued, since the call does not allow the caller to specify a destination address.
3. *writev()* is an atomic operation on sockets of type SOCK_DGRAM and SOCK_RAW in that it

produces one packet of data every time it is issued. For example, a *writev()* to a datagram socket results in a single datagram.

4. To broadcast on an AF_INET socket, the socket option SO_BROADCAST must be set (with a *setsockopt()*).

5. When using a connection-oriented transport service, all errors except [EUNATCH] and [EUNKNOWN] are mapped to [EPIPE] on an output operation when either of the following occurs:
    - A connection that is in progress is unsuccessful.
    - An established connection is broken.

    To get the actual error, use *getsockopt()* with the SO_ERROR option, or perform an input operation (for example, *read()*).
6. For the file systems that do not support large files, **writev()** 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, **writev()** will return [EFBIG] if the starting offset exceeds 2GB minus 2 bytes and the file was not opened for large file access.

7. 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.
8. QOPT File System Differences

    When writing to files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being written are ignored.
9. 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 character special file. In addition, the change and modification times for the file are updated.

10. If the write exceeds the process soft file size limit, signal SIFXFSZ is issued.

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
- 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

API introduced: V3R1

Process a Path Name Exit Program

Required Parameter Group:

1	Selection status pointer	Input	BINARY(4)
2	Error value pointer	Input	BINARY(4)
3	Return value pointer	Output	BINARY(4)
4	Object name pointer	Input	CHAR(*)
5	Function control block pointer	Input	CHAR(*)

The Process a Path Name exit program is a user-specified exit program that is called by the **Qp0lProcessSubtree()** function for each object in the API's search that meets the caller's selection criteria. This exit program can be either a procedure or program.

When the user exit program is given control, it can call other APIs, build lists or tables, or do other processing. Since the API passes the names of all the children objects to the user exit program before passing the name of the parent, the user exit program can also delete directories.

If the exit program encounters an error during processing, it returns a valid errno in the Return value pointer field, that **Qp0lProcessSubtree()** returns to its caller. When its processing is complete, the exit program return code is set to tell **Qp0lProcessSubtree()** to do one of the following:

- End processing.
- Continue processing by calling the exit program again with the next object from the same directory.
- Continue processing by calling the exit program again, but not with objects from the same directory. In this case, **Qp0lProcessSubtree()** moves to the next directory or object that meets the specified criteria and calls the exit program with it.

If **Qp0lProcessSubtree()** encounters any problems in resolving to a user exit program, **Qp0lProcessSubtree()** ends and returns to its caller. If **Qp0lProcessSubtree()** encounters any errors with any other parameters, it ends and returns control to its caller, after a call to the user exit program. This call allows the exit program to perform any desired cleanup before **Qp0lProcessSubtree()** ends. Use the *Err_recovery_action* parameter of **Qp0lProcessSubtree()** to set other conditions for calling or not calling the user exit program.

Storage referred to by the Selection status pointer, Error value pointer, Return value pointer, or the Object name pointer when the Process a Path Name exit program is called, are destroyed or reused when **Qp0lProcessSubtree()** regains control.

See Qp0lProcessSubtree()--Process a Path Name for more information.

Parameters

Selection status pointer

INPUT; BINARY(4)

A pointer to an unsigned integer. This pointer indicates whether **Qp0lProcessSubtree()**

encountered any problems in processing. Valid values follow:

0 ***QP0L_SELECT_OK:*** Indicates to that no problems were encountered during the selection of the current object. The Error value pointer parameter is set to NULL.

1 ***QP0L_SELECT_DONE:*** Indicates that the last object was processed and that this is the last call to the Process a Path Name exit program. The Object name pointer and the Error value pointer parameters are set to NULL.

2 ***QP0L_SELECT_NOT_OK:*** Indicates that **Qp0lProcessSubtree()** has encountered an error but that the Process a Path Name exit program can decide if the operation should continue or end. The Error value pointer parameter points to a valid errno.

3 ***QP0L_SELECT_FAILED:*** Indicates that **Qp0lProcessSubtree()** has encountered an unrecoverable error and that **Qp0lProcessSubtree()** will return to its caller when it regains control. The Error value pointer parameter points to a valid errno.

Error value pointer

INPUT; BINARY(4)

A pointer to a valid errno that describes any problems encountered by the API during the processing of the current object. Any valid errno can be passed in this field when this parameter is not NULL.

Return value pointer

OUTPUT; BINARY(4)

A pointer to a value from the Process a Path Name exit program that instructs the API to continue or to end processing. Valid values follow.

0	**Process a Path Name exit program** was successful.
-1	**Process a Path Name exit program** was successful. **Qp0lProcessSubtree()** should skip processing any remaining objects in this directory and move on to process objects in other directories.
> 0 (an errno)	**Process a Path Name exit program** was not successful. **Qp0lProcessSubtree()** ends.

Object name pointer

INPUT; CHAR(*)

A pointer to the path name structure that contains the fully qualified name of the object being processed by **Qp0lProcessSubtree()**. The Path_Type flag defined in the qlg.h header file must be used to determine whether the Object name pointer contains a pointer or is a character string. This flag must also be used to determine whether the path name delimiter character is 1 or 2 characters long. Value values follow:

0 The path name is a character string, and the path name delimiter is 1 character long.

1 The path name is a pointer, and the path name delimiter character is 1 character long.

2 The path name is a character string, and the path name delimiter is 2 characters long.

3 The path name is a pointer, and the path name delimiter character is 2 characters long.

Function control block pointer

INPUT; CHAR(*)

A pointer to the data that is passed to **Qp0lProcessSubtree()** on its call. **Qp0lProcessSubtree()** does not process the data that is referred to by this pointer, but passes this pointer as a parameter when it calls the exit program.

Exit program introduced: V4R3

Top | UNIX-Type APIs | APIs by category

Save Storage Free Exit Program

Required Parameter Group:			
1	Path name pointers	Input	Char(*)
2	Return code pointer	Output	Binary(4)
3	Return value pointer	Output	Binary(4)
4	Function control block pointer	Input	Char(*)

The Save Storage Free exit program is a user-specified program that is called by **Qp0lSaveStgFree()** to save an OS/400 object of type *STMF. This exit program can be either a procedure or program.

When the Save Storage Free exit program is given control, it should save the object so it can be dynamically retrieved at a later time. The *STMF object is locked when the exit program is called to prevent changes to it until the storage free operation is complete. If the Save Storage Free exit program ends unsuccessfully, it must return a valid *errno* in the storage pointed to by the return value pointer. **Qp0lSaveStgFree()** then passes this *errno* to its caller with a minus one return code.

Storage referred to by the path name pointers or the return code pointer when the Save Storage Free exit program is called is destroyed or reused when **Qp0lSaveStgFree()** regains control.

Required Parameter Group

Path names pointers

INPUT; CHAR(*)

All of the path names to the *STMF object being storage freed. There is one path name for each link to the object. These path names are in the Qlg_Path_Name_T format and are in the UCS-2 CCSID. See Path name format for more information on this format. For information about UCS-2, see the Globalization topic.

Path Name Pointers			
Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Number of path names
4	4	CHAR(12)	Reserved
16	10	ARRAY(*)	Array of path name pointers

Array of path name pointers. Pointers to each path name that **Qp0lSaveStgFree()** found for the object identified by the path name on the call to **Qp0lSaveStgFree()**. Each path name is in the Qlg_Path_Name_T format.

Number of path names. The total number of path names that **Qp0lSaveStgFree()** found for the object identified by the caller of **Qp0lSaveStgFree()**.

Reserved. A reserved field. This field must be set to binary zero.

Return code pointer

OUTPUT; BINARY(4)

A pointer to an indicator that is returned to indicate whether the exit program was successful or whether it failed. Valid values follow:

0 The Save Storage Free exit program was successful.

-1 The Save Storage Free exit program was not successful. The Return value pointer is set to indicate the error.

Return value pointer

OUTPUT; BINARY(4)

A pointer to a valid *errno* that is returned from the exit program to identify the reason it was not successful.

Function control block pointer

INPUT; CHAR(*)

A pointer to the data that is passed to **Qp0lSaveStgFree()** on its call. **Qp0lSaveStgFree()** does not process the data that is referred to by this pointer, but passes this pointer as a parameter when it calls the exit program.

Related Information

- Qp0lSaveStgFree()--Save Storage Free

Exit program introduced: V4R3

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.

Time Stamp Updates for Integrated File System APIs			
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
qp0lror.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.
ENOPROTOOPT	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.
EOVERFLOW	3525	Object is too large to process.
EJRNDAMAGE	3526	Journal is damaged.
EJRNINACTIVE	3527	Journal is inactive.
EJRNRCVSPC	3528	Journal space or system storage error.
EJRNRMT	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.