# UNIX-Type APIs (V5R2)

## Process-Related APIs

---

## Table of Contents


Process-Related APIs

- APIs
  - getopt() (Get flag letters from argument vector)
  - getpgrp() (Get process group ID)
  - getpid() (Get process ID)
  - getppid() (Get process ID of parent process)
  - getrlimit() (Get resource limit)
  - pipe() (Create interprocess channel)
  - QlgSpawn() (Spawn process (using NLS-enabled path name))
  - QlgSpawnp() (Spawn process with path (using NLS-enabled file name))
  - Qp0wChkChld() (Check status for child processes)
  - Qp0wChkPgrp() (Check status for process group)
  - Qp0wChkPid() (Check status for process ID)
  - Qp0wGetJobID() (Get qualified job name and ID for process ID)
  - Qp0wGetPgrp() (Get process group ID)
  - Qp0wGetPid() (Get process ID)
  - Qp0wGetPidNoInit (Get process ID without initializing for signals)
  - Qp0wGetPPid() (Get process ID of parent process)
  - Qp0zPipe() (Create interprocess channel with sockets)
  - Qp0zSystem() (Run a CL command)
  - setpgid() (Set process group ID for job control)
  - setrlimit() (Set resource limit)
  - spawn() (Spawn process)
  - spawnp() (Spawn process with path)
  - ulimit() (Get and set process limits)
  - wait() (Wait for child process to end)
  - waitpid() (Wait for specific child process)
- About shell scripts

Header Files for UNIX-Type Functions
Errno Values for UNIX-Type Functions

# Process-Related APIs

The process-related APIs perform process-related or other general operations. These APIs are C language functions that can be used in ILE C programs.

The process-related APIs are:

- getopt() (Get flag letters from argument vector) returns the next flag letter in the argv list that matches a letter in optionstring.
- getpgrp() (Get process group ID) returns the process group ID of the calling process.
- getpid() (Get process ID) returns the process ID of the calling process.
- getppid() (Get process ID of parent process) returns the parent process ID of the calling process.
- getrlimit() (Get resource limit) returns the resource limit for the specified *resource*.
- pipe() (Create interprocess channel) creates a data pipe and places two file descriptors, one each into the arguments fildes[0] and fildes[1], that refer to the open file descriptions for the read and write ends of the pipe, respectively.
- QlgSpawn() (Spawn process (using NLS-enabled path name)) creates a child process that inherits specific attributes from the parent.
- QlgSpawnp() (Spawn process with path (using NLS-enabled file name)) creates a child process that inherits specific attributes from the parent.
- Qp0wChkChld() (Check status for child processes) returns the status and process table entry information for the child processes of the specified process ID.
- Qp0wChkPgrp() (Check status for process group) returns the status and process table entry information for the processes that are members of the process group identified by pid in the structure QP0W_PID_Entry_T.
- Qp0wChkPid() (Check status for process ID) returns the status and process table entry information for the process specified by the process ID pid.
- Qp0wGetJobID() (Get qualified job name and ID for process ID) returns the qualified job name and internal job identifier for the process whose process ID matches pid.
- Qp0wGetPgrp() (Get process group ID) returns the process group ID of the calling process.
- Qp0wGetPid() (Get process ID) returns the process ID of the calling process.
- Qp0wGetPidNoInit (Get process ID without initializing for signals) returns the process ID of the calling process without enabling the process to receive signals.
- Qp0wGetPPid() (Get process ID of parent process) returns the parent process ID of the calling process.
- Qp0zPipe() (Create interprocess channel with sockets) creates a data pipe that can be used by two processes.
- Qp0zSystem() (Run a CL command) spawns a new process, passes CLcommand to the CL command processor in the new process, and waits for the command to complete.
- setpgid() (Set process group ID for job control) is used to either join an existing process group or create a new process group within the session of the calling process.
- setrlimit() (Set resource limit) sets the resource limit for the specified *resource*.
- spawn() (Spawn process) creates a child process that inherits specific attributes from the parent.

- spawnp() (Spawn process with path) creates a child process that inherits specific attributes from the parent.
- »ulimit() (Get and set process limits) provides a way to get and set process resource limits.«
- wait() (Wait for child process to end) suspends processing until a child process has ended.
- waitpid() (Wait for specific child process) allows the calling thread to obtain status information for one of its child processes.

For additional information, see About shell scripts.

---

# getopt()--Get Flag Letters from Argument Vector

Syntax

```
#include <unistd.h>

int getopt(int argc, char * const argv[],
     const char *optionstring);
```

Service Program Name: QP0ZCPA

Default Public Authority: *USE

Threadsafe: No

The **getopt()** function returns the next flag letter in the argv list that matches a letter in optionstring. The optarg external variable is set to point to the start of the flag's parameter on return from **getopt()**

**getopt()** places the argv index of the next argument to be processed in optind. The optind variable is external. It is initialized to 1 before the first call to **getopt()**.

**getopt()** can be used to help a program interpret command line flags that are passed to it.

## Parameters

***argc***

(Input) The number of parameters passed by the function.

***argv***

(Input) The list of parameters passed to the function.

***optionstring***

(Input) A string of flag letters. The string must contain the flag letters that the program using **getopt()** recognizes. If a letter is followed by a colon, the flag is expected to have an argument or group of arguments, which can be separated from it by blank spaces.

The special flag "--" (two hyphens) can be used to delimit the end of the options. When this flag is encountered, the "--" is skipped and EOF is returned. This flag is useful in delimiting arguments beginning with a hyphen that are not options.

## Authorities

None.

## Return Value

| | |
|---|---|
| EOF | **getopt()** processed all flags (that is, up to the first argument that is not a flag). |
| '?' | **getopt()** encountered a flag letter that was not included in optionstring. The variable optopt is set to the real option found in argv regardless of whether the flag is in optionstring of not. An error message is printed to `stderr`. The generation of error messages can be suppressed by setting opterr to 0. |

## Error Conditions

The **getopt()** function does not return an error.

## Example

See Code disclaimer information for information pertaining to code examples.

The following example processes the flags for a command that can take the mutually exclusive flags a and b, and the flags f and o, both of which require parameters.

```
#include <unistd.h>

int main( int argc, char *argv[] )
{
 int c;
 extern int optind;
 extern char *optarg;
 .
 .
 .
 while ((c = getopt(argc, argv, "abf:o:")) != EOF)
 {
  switch (c)
  {
   case 'a':
    if (bflg)
     errflg++;
    else
     aflg++;
    break;
   case 'b':
    if (aflg)
     errflg++;
    else
```

```
      bflg++;
     break;
    case 'f':
     ifile = optarg;
     break;
    case 'o':
     ofile = optarg;
     break;
    case '?':
     errflg++;
   } /* case */
   if (errflg)
   {
    fprintf(stderr, "usage: . . . ");
    exit(2);
   }
  } /* while */
  for ( ; optind < argc; optind++)
  {
   if (access(argv[optind], R_OK))
   {
    .
    .
    .
   }
  } /* for */
}    /* main */
```

---

API introduced: V3R6

---

# getpgrp()--Get Process Group ID

Syntax

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

pid_t getpgrp(void);
```

Service Program Name: QP0WSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **getpgrp()** function returns the process group ID of the calling process.

## Parameters

None

## Authorities

None.

## Return Value

*pid_t* The value returned by **getpgrp()** is the process group ID of the calling process.

## Error Conditions

The **getpgrp()** function is always successful and does not return an error.

## Usage Notes

The **getpgrp()** function enables a process for signals if the process is not already enabled for signals. For details, see Qp0sEnableSignals()--Enable Process for Signals.

## Related Information

- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- Qp0wGetPgrp()--Get Process Group ID

## Example

For an example of using this function, see the child program in Using the Spawn Process and wait for Child Process APIs in Appendix A, Examples.

---

API introduced: V3R6

---

# getpid()--Get Process ID

Syntax

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

pid_t getpid(void);
```

Service Program Name: QP0WSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **getpid()** function returns the process ID of the calling process.

## Parameters

None

## Authorities

None.

## Return Value

| | |
|---|---|
| *pid_t* | The value returned by **getpid()** is the process ID of the calling process. |

## Error Conditions

The **getpid()** function is always successful and does not return an error.

## Usage Notes

The **getpid()** function enables a process for signals if the process is not already enabled for signals. For details, see Qp0sEnableSignals()--Enable Process for Signals.

## Related Information

- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- Qp0sDisableSignals()--Disable Process for Signals
- Qp0sEnableSignals()--Enable Process for Signals
- Qp0wGetPid()--Get Process ID
- Qp0wGetPidNoInit()--Get Process ID without Initializing for Signals

## Example

For an example of using this function, see the child program in Using the Spawn Process and Wait for the Child Process APIs in Appendix A, Examples.

---

API introduced: V3R6

---

# getppid()--Get Process ID of Parent Process

Syntax

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

pid_t getppid(void);
```

Service Program Name: QP0WSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **getppid()** function returns the parent process ID of the calling process.

## Parameters

None

## Authorities

None.

## Return Value

| | |
|---|---|
| *pid_t* | The value returned by **getppid()** is the process ID of the parent process for the calling process. A process ID value of 1 indicates that there is no parent process associated with the calling process. |

## Error Conditions

The **getppid()** function is always successful and does not return an error.

## Related Information

- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)

- Qp0wGetPPid()--Get Process ID of Parent Process

## Example

For an example of using this function, see the child program in Using the Spawn Process and Wait for Child Process APIs in Appendix A, Examples.

---

API introduced: V3R6

---

# »getrlimit()--Get resource limit

Syntax

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

int getrlimit(int resource, struct rlimit *rlp);
```

Service Program Name: QP0WSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **getrlimit()** function returns the resource limit for the specified *resource*. A resource limit is a way for the operating system to enforce a limit on a variety of resources used by a process. A resource limit is represented by a rlimit structure. The rlim_cur member specifies the current or soft limit and the rlim_max member specifies the maximum or hard limit.

The **getrlimit()** function supports the following resources:

| | |
|---|---|
| *RLIMIT_FSIZE (0)* | The maximum size of a file in bytes that can be created by a process. |
| *RLIMIT_NOFILE (1)* | The maximum number of file descriptors that can be opened by a process. |
| *RLIMIT_CORE (2)* | The maximum size of a core file in bytes that can be created by a process. |
| *RLIMIT_CPU (3)* | The maximum amount of CPU time in seconds that can be used by a process. |
| *RLIMIT_DATA (4)* | The maximum size of a process' data segment in bytes. |
| *RLIMIT_STACK (5)* | The maximum size of a process' stack in bytes. |
| *RLIMIT_AS (6)* | The maximum size of a process' total available storage in bytes. |

The value of RLIM_INFINITY is considered to be larger than any other limit value. If the value of the limit is RLIM_INFINITY, then a limit is not enforced for that resource. The **getrlimit()** function always returns RLIM_INFINITY for the following resources: RLIMIT_AS, RLIMIT_CORE, RLIMIT_CPU, RLIMIT_DATA, and RLIMIT_STACK.

## Parameters

***resource***

(Input)

The resource to get the limits for.

***rlp**

(Output)

Pointer to a struct rlim_t where the values of the hard and soft limits are returned.

## Authorities and Locks

None.

## Return Value

**0** **getrlimit()** was successful.

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

## Error Conditions

If **getrlimit()** 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] An invalid parameter was found.

An invalid *resource* was specified.

## Related Information

- The <**sys/resource.h**> file (see Header Files for UNIX-Type Functions)
- setrlimit()-Set resource limit
- ulimit()-Get and set process limits

## Example

```
#include <sys/resource.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main (int argc, char *argv[])
{
```

```
  struct rlimit limit;

  /* Set the file size resource limit. */
  limit.rlim_cur = 65535;
  limit.rlim_max = 65535;
  if (setrlimit(RLIMIT_FSIZE, &limit) != 0) {
    printf("setrlimit() failed with errno=%d\n", errno);
    exit(1);
  }

  /* Get the file size resource limit. */
  if (getrlimit(RLIMIT_FSIZE, &limit) != 0) {
    printf("getrlimit() failed with errno=%d\n", errno);
    exit(1);
  }

  printf("The soft limit is %llu\n", limit.rlim_cur);
  printf("The hard limit is %llu\n", limit.rlim_max);
  exit(0);
}
```

**Example Output:**

```
The soft limit is 65535
The hard limit is 65535
```

«

---

Introduced: V5R2

---

# pipe()--Create an Interprocess Channel

Syntax

```
#include <unistd.h>

int pipe(int fildes[2]);
```

Service Program Name: QP0LLIB1

Default Public Authority: *USE

Threadsafe: Yes

The **pipe()** function creates a data pipe and places two file descriptors, one each into the arguments *fildes[0]* and *fildes[1]*, that refer to the open file descriptions for the read and write ends of the pipe, respectively. Their integer values will be the two lowest available at the time of the **pipe()** call. The O_NONBLOCK and FD_CLOEXEC flags will be clear on both descriptors. NOTE: these flags can, however, be set by the **fcntl()** function.

Data can be written to the file descriptor *fildes[1]* and read from file descriptor *fildes[0]*. A read on the file descriptor *fildes[0]* will access data written to the file descriptor *fildes[1]* on a first-in-first-out basis. File descriptor *fildes[0]* is open for reading only. File descriptor *fildes[1]* is open for writing only.

The **pipe()** function is often used with the **spawn()** function to allow the parent and child processes to send data to each other.

Upon successful completion, **pipe**() will update the access time, change time, and modification time of the pipe.

## Parameters

***fildes[2]***
(Output) An integer array of size 2 that will receive the pipe descriptors.

## Authorities

None.

## Return Value

*0* **pipe**() was successful.

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

## Error Conditions

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

*[EFAULT]*
: The address used for an argument is not correct.

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

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

*[EMFILE]*
: Too many open files for this process.

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

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

*[ENFILE]*
: Too many open files in the system.

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

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

*[ENOMEM]*
: Storage allocation request failed.

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

  There is not enough memory to perform the requested function.

*[EUNKNOWN]*
: Unknown system state.

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

## Related Information

- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- The <**fcntl.h**> file (see Header Files for UNIX-Type Functions)
- fcntl()--Perform File Control Command
- fstat()--Get File Information by Descriptor
- Qp0zPipe()--Create Interprocess Channel with Sockets
- read()--Read from Descriptor
- spawn()--Spawn Process
- write()--Write to Descriptor

## Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a pipe, writes 10 bytes of data to the pipe, and then reads those 10 bytes of data from the pipe.

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

void main()
{
   int fildes[2];
   int rc;
   char writeData[10];
   char readData[10];
   int bytesWritten;
   int bytesRead;

   memset(writeData,'A',10);

   if (-1 == pipe(fildes))
   {
      perror("pipe error");
      return;
   }

   if (-1 == (bytesWritten = write(fildes[1],
                                    writeData,
                                    10)))
   {
      perror("write error");
   }
   else
   {
      printf("wrote %d bytes\n",bytesWritten);

      if (-1 == (bytesRead = read(fildes[0],
                                   readData,
                                   10)))
      {
         perror("read error");
      }
      else
      {
         printf("read %d bytes\n",bytesRead);
      }
   }

   close(fildes[0]);
   close(fildes[1]);
```

```
    return;
}
```

---

API introduced: V5R1

---

Top | UNIX-Type APIs | APIs by category

# QlgSpawn()--Spawn Process (using NLS-enabled path name)

Syntax

```
#include <spawn.h>
#include <qlg.h>

pid_t QlgSpawn(const Qlg_Path_Name_T     *path,
               const int                 fd_count,
               const int                 fd_map[],
               const struct inheritance  *inherit,
               char * const              argv[],
               char * const              envp[]);
```

Service Program Name: QP0ZSPWN

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgSpawn()** function, like the **spawn()** function, creates a child process that inherits specific attributes from the parent. The difference is that for the *path* parameter, the **QlgSpawn()** function takes a pointer to a Qlg_Path_Name_T structure, while the **spawn()** function takes a pointer to a character string in the CCSID of the job.

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, and return values, see spawn()--Spawn Process.

## Parameters

***path***

(Input) A pointer to a Qlg_Path_Name_T structure that contains a specific path name or a pointer to a specific path name of an executable file that will run in the new (child) process. For more information on the Qlg_Path_Name_T structure, see Path name format.

## Usage Notes

See spawn()--Spawn Process for a complete discussion of usage information for **QlgSpawn()**. In addition, the following should be noted specifically for **QlgSpawn()**.

1. Shell scripts are supported; however, the interpreter path in the shell script itself cannot be a Qlg_Path_Name_T structure.

## Related Information

- The <**qlg.h**> file (see Header Files for UNIX-Type Functions)
- spawn()--Spawn Process
- QlgSpawnp()--Spawn Process with Path (using NLS-enabled file name)

## Example

See Code disclaimer information for information pertaining to code examples.

**Parent Program**

The following ILE C for OS/400 program can be created in any library. This parent program assumes the corresponding child program will be created with the name CHILD in the library QGPL. Call this parent program with no parameters to run the example.

```
/*******************************************************************/
/*******************************************************************/
/*                                                                 */
/* FUNCTION:  This program acts as a parent to a child program.   */
/*                                                                 */
/* LANGUAGE:  ILE C for OS/400                                     */
/*                                                                 */
/* APIs USED: QlgSpawn(), waitpid(),                               */
/*            QlgCreat(), QlgUnlink(), QlgOpen()                   */
/*                                                                 */
/*******************************************************************/
/*******************************************************************/
#include <errno.h>
#include <fcntl.h>
#include <spawn.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <qlg.h>
#include <Qp0lstdi.h>

#define ARGV_NUM   6
#define ENVP_NUM   1
#define CHILD_PGM  "QGPL/CHILD"
#define spwpath "/QSYS.LIB/QGPL.LIB/CHILD.PGM"
#define fpath "A_File"

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

/* This is a parent program that will use QlgSpawn() to start a     */
/* child.  A file is created that is written to, both by the parent */
/* and the child.  The end result of the file will look something   */
/* like the following:                                              */
/*     Parent writes    Child writes                                */
/*     -------------    ---------------------------------------     */
/*           1          argv[0]  getppid()  getpgrp()  getpid()     */
/* The parent uses waitpid() to wait for the child to return and to */
/* retrieve the resulting status of the child when it does return.  */

int main(int argc, char *argv[])
{
    int    rc;                     /* API return code */
    int    fd, fd_read;            /* parent file descriptors */
    char   fd_str[4];              /* file descriptor string */
    const char US_const[3]= "US";
    const char Language_const[4]="ENU";
    const char Path_Name_Del_const[2]= "/";
    struct pnstruct f_path_name;   /* file pathname */
    int    buf_int;                /* write(), read() buffer */
    char   buf_pgm_name[22];       /* read() program name buffer */
```

```
struct pnstruct spw_path;      /* QlgSpawn() *path */
int    spw_fd_count = 0;       /* QlgSpawn() fd_count */
struct inheritance spw_inherit; /* QlgSpawn() *inherit */
char  *spw_argv[ARGV_NUM];     /* QlgSpawn() *argv[] */
char  *spw_envp[ENVP_NUM];     /* QlgSpawn() *envp[] */
int    seq_num;                /* sequence number */
char   seq_num_str[4];         /* sequence number string */
pid_t  pid;                    /* parent pid */
char   pid_str[11];            /* parent pid string */
pid_t  pgrp;                   /* parent process group */
char   pgrp_str[11];           /* parent process group string */
pid_t  spw_child_pid;          /* QlgSpawn() child pid */
pid_t  wt_child_pid;           /* waitpid() child pid */
int    wt_stat_loc;            /* waitpid() *stat_loc */
int    wt_pid_opt = 0;         /* waitpid() option */

/* Get the pid and pgrp for the parent. */
pid = getpid();
pgrp = getpgrp();
/* Format the pid and pgrp value into null-terminated strings. */
sprintf(pid_str, "%d", pid);
sprintf(pgrp_str, "%d", pgrp);

/* Initialize Qlg_Path_Name_T parameters */
memset(&f_path_name,0x00,sizeof(struct pnstruct));
f_path_name.qlg_struct.CCSID = 37;
memcpy(f_path_name.qlg_struct.Country_ID,US_const,2);
memcpy(f_path_name.qlg_struct.Language_ID,Language_const,3);
f_path_name.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
f_path_name.qlg_struct.Path_Length = sizeof(fpath)-1;
memcpy(f_path_name.qlg_struct.Path_Name_Delimiter,
       Path_Name_Del_const,1);
memcpy(f_path_name.pn,fpath,sizeof(fpath)-1);

/* Create a file and maintain the file descriptor. */
fd = QlgCreat((Qlg_Path_Name_T *)&f_path_name, S_IRWXU);
if (fd == -1)
  {
    printf("FAILURE: QlgCreat() with errno = %d\n",errno);
    return -1;
  }
/* Format the file descriptor into null-terminated string. */
sprintf(fd_str, "%d", fd);

/* Initialize Qlg_Path_Name_T parameters */
memset(&spw_path,0x00,sizeof(struct pnstruct));
spw_path.qlg_struct.CCSID = 37;
memcpy(spw_path.qlg_struct.Country_ID,US_const,2);
memcpy(spw_path.qlg_struct.Language_ID,Language_const,3);
spw_path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
spw_path.qlg_struct.Path_Length = sizeof(spwpath)-1;
memcpy(spw_path.qlg_struct.Path_Name_Delimiter,
       Path_Name_Del_const,1);
memcpy(spw_path.pn,spwpath,sizeof(spwpath)-1);

/* Write a '1' out to the file. */
seq_num = 1;
sprintf(seq_num_str, "%d", seq_num);
buf_int = seq_num;
write(fd, &buf_int, sizeof(int));

/* Set the QlgSpawn() child arguments for the child.     */
/* NOTE: The child will always get argv[0] in the        */
/* LIBRARY/PROGRAM notation, but the QlgSpawn() argv[0]  */
/* (spw_argv[0] in this case) must be non-NULL in order  */
/* to allow additional arguments.  For this example, the */
/* CHILD_PGM was chosen.                                 */
/* NOTE: The parent pid and the parent process group are */
/* passed to the child for demonstration purposes only.  */
spw_argv[0] = CHILD_PGM;
```

```
spw_argv[1] = pid_str;
spw_argv[2] = pgrp_str;
spw_argv[3] = seq_num_str;
spw_argv[4] = fd_str;
spw_argv[5] = NULL;

/* This QlgSpawn() will use simple inheritance for file  */
/* descriptors (fd_map[] value is NULL).                 */
memset(&spw_inherit,0x00,sizeof(spw_inherit));
spw_envp[0] = NULL;
spw_child_pid = QlgSpawn((Qlg_Path_Name_T *)&spw_path,
                spw_fd_count, NULL, &spw_inherit, spw_argv, spw_envp);
if (spw_child_pid == -1)
  {
    printf("FAILURE: QlgSpawn() with errno = %d\n",errno);
    close(fd);
    QlgUnlink((Qlg_Path_Name_T *)&f_path_name);
    return -1;
  }

/* The parent no longer needs to use the file descriptor, so */
/* it can close it, now that it has issued QlgSpawn().       */
rc = close(fd);
if (rc != 0)
    printf("FAILURE: close(fd) with errno = %d\n",errno);

/* NOTE: The parent can continue processing while the child is */
/* also processing.  In this example, though, the parent will  */
/* simply wait until the child finishes processing.            */
/* Issue waitpid() in order to wait for the child to return.   */
wt_child_pid = waitpid(spw_child_pid,&wt_stat_loc,wt_pid_opt);
if (wt_child_pid == -1)
  {
    printf("FAILURE: waitpid() with errno = %d\n",errno);
    close(fd);
    QlgUnlink((Qlg_Path_Name_T *)&f_path_name);
    return -1;
  }

/* Check to ensure the child did not encounter an error  */
/* condition.                                            */
if (WIFEXITED(wt_stat_loc))
  {
    if (WEXITSTATUS(wt_stat_loc) != 1)
        printf("FAILURE: waitpid() exit status = %d\n",
               WEXITSTATUS(wt_stat_loc));
  }
else
    printf("FAILURE: unknown child status\n");

/* Open the file for read to verify what the child wrote. */
fd_read = QlgOpen((Qlg_Path_Name_T *)&f_path_name, O_RDONLY);
if (fd_read == -1)
  {
    printf("FAILURE: open() for read with errno = %d\n",errno);
    QlgUnlink((Qlg_Path_Name_T *)&f_path_name);
    return -1;
  }

/* Verify what child wrote. */
rc = read(fd_read, &buf_int, sizeof(int));
if ( (rc != sizeof(int)) || (buf_int != 1) )
    printf("FAILURE: read()\n");
memset(buf_pgm_name,0x00,sizeof(buf_pgm_name));
rc = read(fd_read, buf_pgm_name, strlen(CHILD_PGM));
if ( (rc != strlen(CHILD_PGM)) ||
     (strcmp(buf_pgm_name,CHILD_PGM) != 0) )
    printf("FAILURE: read() child argv[0]\n");
rc = read(fd_read, &buf_int, sizeof(int));
if ( (rc != sizeof(int)) || (buf_int != pid) )
```

```
        printf("FAILURE: read() child getppid()\n");
    rc = read(fd_read, &buf_int, sizeof(int));
    if ( (rc != sizeof(int)) || (buf_int != pgrp) )
        printf("FAILURE: read() child getpgrp()\n");
    rc = read(fd_read, &buf_int, sizeof(int));
    if ( (rc != sizeof(int)) || (buf_int != spw_child_pid) ||
         (buf_int != wt_child_pid) )
        printf("FAILURE: read() child getpid()\n");

    /* Attempt one more read() to ensure there is no more data. */
    rc = read(fd_read, &buf_int, sizeof(int));
    if (rc != 0)
        printf("FAILURE: read() past end of data\n");

    /* The parent no longer needs to use the read() file descriptor, */
    /* so it can close it.                                            */
    rc = close(fd_read);
    if (rc != 0)
        printf("FAILURE: close(fd_read) with errno = %d\n",errno);

    /* Clean up by performing unlink(). */
    rc = QlgUnlink((Qlg_Path_Name_T *)&f_path_name);
    if (rc != 0)
      {
        printf("FAILURE: QlgUnlink() with errno = %d\n",errno);
        return -1;
      }
    printf("completed successfully\n");
    return 0;
}
```

**Child Program**

The following ILE C for OS/400 program must be created with the name CHILD in the library QGPL in order to be found by the parent program. This program is not to be called directly, as it is run through the use of QlgSpawn() in the parent program.

```
/*******************************************************************/
/*******************************************************************/
/*                                                                 */
/* FUNCTION:  This program acts as a child to a parent program.    */
/*                                                                 */
/* LANGUAGE:  ILE C for OS/400                                     */
/*                                                                 */
/* APIs USED: getpid(), getppid(), getpgrp()                       */
/*                                                                 */
/*******************************************************************/
/*******************************************************************/
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

/* This is a child program that gets control from a parent program */
/* that issues QlgSpawn().  This particular child program expects  */
/* the following 5 arguments (all are null-terminated strings):    */
/*     argv[0] - child program name                                */
/*     argv[1] - parent pid (for demonstration only)               */
/*     argv[2] - parent process group (for demonstration only)     */
/*     argv[3] - sequence number                                   */
/*     argv[4] - parent file descriptor                            */
/* If the child program encounters an error, it returns with a     */
/* value greater than 50.  If the parent uses wait() or waitpid(), */
/* this return value can be interrogated using the WIFEXITED and   */
/* WEXITSTATUS macros on the resulting wait() or waitpid()         */
/* *stat_loc field.                                                */
```

```
int main(int argc, char *argv[])
{
    pid_t  p_pid;                 /* parent pid argv[1] */
    pid_t  p_pgrp;                /* parent process group argv[2] */
    int    seq_num;               /* parent sequence num argv[3] */
    int    fd;                    /* parent file desc argv[4] */
    int    rc;                    /* API return code */
    pid_t  pid;                   /* getpid() - child pid */
    pid_t  ppid;                  /* getppid() - parent pid */
    pid_t  pgrp;                  /* getpgrp() - process group */

    /* Get the pid, ppid, and pgrp for the child. */
    pid = getpid();
    ppid = getppid();
    pgrp = getpgrp();

    /* Verify 5 parameters were passed to the child. */
    if (argc != 5)
        return 60;

    /* Since the parameters passed to the child using QlgSpawn() are */
    /* pointers to strings, convert the parent pid, parent process   */
    /* group, sequence number, and the file descriptor from strings  */
    /* to integers.                                                  */
    p_pid = atoi(argv[1]);
    p_pgrp = atoi(argv[2]);
    seq_num = atoi(argv[3]);
    fd = atoi(argv[4]);

    /* Verify the getpid() value of the parent is the same as the    */
    /* getppid() value of the child.                                 */
    if (p_pid != ppid)
        return 61;

    /* If the sequence number is 1, simple inheritance was used in   */
    /* this case.  First, verify the getpgrp() value of the parent   */
    /* is the same as the getpgrp() value of the child.  Next, the   */
    /* child will use the file descriptor passed in to write the     */
    /* child's values for argv[0], getppid(), getpgrp(),             */
    /* and getpid().  Finally, the child returns, which will satisfy */
    /* the parent's wait() or waitpid().                             */
    if (seq_num == 1)
      {
        if (p_pgrp != pgrp)
            return 70;
        rc = write(fd, argv[0], strlen(argv[0]));
        if (rc != strlen(argv[0]))
            return 71;
        rc = write(fd, &ppid, sizeof(pid_t));
        if (rc != sizeof(pid_t))
            return 72;
        rc = write(fd, &pgrp, sizeof(pid_t));
        if (rc != sizeof(pid_t))
            return 73;
        rc = write(fd, &pid, sizeof(pid_t));
        if (rc != sizeof(pid_t))
            return 74;
        return seq_num;
      }

    /* If the sequence number is an unexpected value, return */
    /* indicating an error.                                  */
    else
        return 99;
}
```

---

API introduced: V5R1

# QlgSpawnp()--Spawn Process with Path (using NLS-enabled file name)

Syntax

```
#include <spawn.h>
#include <qlg.h>

pid_t QlgSpawnp(const Qlg_Path_Name_T      *file,
                const int                  fd_count,
                const int                  fd_map[],
                const struct inheritance   *inherit,
                char * const               argv[],
                char * const               envp[]);
```

Service Program Name: QP0ZSPWN

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **QlgSpawnp()** function, like the **spawnp()** function, creates a child process that inherits specific attributes from the parent. The difference is that for the *file* parameter, the **QlgSpawnp()** function takes a pointer to a Qlg_Path_Name_T structure, while the **spawnp()** function takes a pointer to a character string in the ccsid of the job.

Limited information on the *file* parameter is provided here. For more information on the *file* parameter and for a discussion of other parameters, authorities required, and return values, see spawnp()--Spawn Process with Path.

## Parameters

***file***

(Input) A pointer to a Qlg_Path_Name_T structure that contains a file name or a pointer to a file name that is used with the search path to find an executable file that will run in the new (child) process. For more information on the Qlg_Path_Name_T structure, see Path name format.

## Usage Notes

See spawnp()--Spawn Process with Path for a complete discussion of usage information for **QlgSpawnp()**. In addition, the following should be noted specifically for **QlgSpawnp()**.

1. The PATH environment variable is used; however, the PATH environment variable cannot be a Qlg_Path_Name_T structure.

2. Shell scripts are supported; however, the interpreter path in the shell script itself cannot be a Qlg_Path_Name_T structure.

## Related Information

- The <**qlg.h**> file (see Header Files for UNIX-Type Functions)
- spawnp()--Spawn Process with Path
- QlgSpawn()--Spawn Process (using NLS-enabled path name)

**Note:** All of the related information for **spawnp()** applies to **QlgSpawn**().

## Example

For an example of using this function, see the example in the QlgSpawn--Spawn Process (using NLS-enabled path name) API.

---

API introduced: V5R1

---

# Qp0wChkChld()--Check Status for Child Processes

Syntax

```
#include <qp0wpid.h>

int Qp0wChkChld(QP0W_PID_Entries_T *chldinfo);
```

Service Program Name: QP0WPID

Default Public Authority: *USE

Threadsafe: Yes

The **Qp0wChkChld()** function returns the status and process table entry information for the child processes of the specified process ID.

## Parameters

***chldinfo***

(I/O) A pointer to the QP0W_PID_Entry_T structure. This structure contains the process table entry information for the children processes identified by pid.

The structure QP0W_PID_Entry_T is defined in the **<qp0wpid.h>** header file as follows:

```
typedef struct QP0W_PID_Entries_T {
    int           entries_prov;
    int           entries_could;
    int           entries_return;
    pid_t         pid;
    QP0W_PID_Data_T  entry[1];
} QP0W_PID_Entries_T;
```

The members of the QP0W_PID_Entry_T structure are as follows:

| | |
|---|---|
| *int entries_prov;* | (Input) The number of entries of type QP0W_PID_Data_T for which that the caller has allocated storage to contain the status and process table entry information. |
| *int entries_could;* | (Output) The number of entries of type QP0W_PID_Data_T that could be returned. If the entries_could value exceeds the entries_prov value, the **Qp0wChkChld()** function should be called again with sufficient storage to contain the number of entries returned in entries_could (entries_prov must be greater than or equal to entries_could). |

| | |
|---|---|
| *int entries_return;* | (Output) The number of entries of type QP0W_PID_Data_T that were returned. If the entries_return value is less than the entries_prov value, the content of the excess number of entries provided is unchanged by **Qp0wChkChld**(). |
| *pid_t pid;* | (Input) The process ID of the process for which information about its child processes is to be returned. |
| *QP0W_PID_Data_T entry[1];* | (Output) The process table information for child processes. There is one QP0W_PID_Data_T structure entry for each child process, limited by the value of entries_prov. |

The structure QP0W_PID_Data_T is defined in the **<qp0wpid.h>** header file as follows:

```
typedef struct QP0W_PID_Data_T {
    pid_t          pid;
    pid_t          ppid;
    pid_t          pgrp;
    int            status;
    unsigned int   exit_status;
} QP0W_PID_Data_T;
```

The members of the QP0W_PID_Data_T structure are as follows:

| | |
|---|---|
| *pid_t pid;* | The process ID of the process. |
| *pid_t ppid;* | The process ID of the parent process. If ppid has a value of binary 1, there is no parent process associated with the process. |
| *pid_t pgrp;* | The process group ID of the process. |
| *int status;* | A collection of flag bits that describe the current state of the process. The following flag bits can be set in status: |

| | |
|---|---|
| *QP0W_PID_TERMINATED* | The process has ended. |
| *QP0W_PID_STOPPED* | The process has been stopped by a signal. |
| *QP0W_PID_CHILDWAIT* | The process is waiting for a child process to be ended or stopped by a signal. |
| *QP0W_PID_SIGNALSTOP* | The process has requested that the SIGCHLD signal be generated for the process when one of its child processes has been stopped by a signal. |

| | |
|---|---|
| *unsigned int exit_status;* | Exit status of the process. This member only has meaning if the status has been set to QP0W_PID_TERMINATED. Refer to the **wait**() function for a description of the exit status for a process. |

## Authorities

The process calling **Qp0wChkChld()** must have the appropriate authority to the process being examined. A process is allowed to examine the process table information for a process if at least one of the following conditions is true:

- The process is calling **Qp0wChkChld()** for its own process.
- The process has *JOBCTL special authority defined in the process user profile or in a current adopted user profile.
- The process is the parent of the process (the process being examined has a parent process ID equal to the process ID of the process calling **Qp0wChkChld**()).
- The real or effective user ID of the process matches the real or effective user ID of the process calling **Qp0wChkChld**().

## Return Value

*0* **Qp0wChkChld**() was successful.

*value* **Qp0wChkChld()** was not successful. The value returned indicates one of the following errors. Under some conditions, *value* could indicate an error other than those listed here.

*[EINVAL]* An invalid parameter was found.

A parameter passed to this function is not valid.

*[EPERM]* Operation not permitted.

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

*[ESRCH]* No item could be found that matches the specified value.

## Usage Notes

The **Qp0wChkChld()** function provides an OS/400-specific way to obtain the process table information for the child processes of the specified process.

## Related Information

- The <**qp0wpid.h**> file (see Header Files for UNIX-Type Functions)
- The <**signal.h**> file (see Header Files for UNIX-Type Functions)

- getpgrp()--Get Process Group ID
- getpid()--Get Process ID
- getppid()--Get Process ID of Parent Process
- Qp0wGetPgrp()--Get Process Group ID
- Qp0wGetPid()--Get Process ID
- Qp0wGetPPid()--Get Process ID of Parent Process
- wait()--Wait for Child Process to End

---

API introduced: V3R6

---

# Qp0wChkPgrp()--Check Status for Process Group

Syntax

```
#include <qp0wpid.h>

int Qp0wChkPgrp(QP0W_PID_Entries_T *mbrinfo);
```

Service Program Name: QP0WPID

Default Public Authority: *USE

Threadsafe: Yes

The **Qp0wChkPgrp()** function returns the status and process table entry information for the processes that are members of the process group identified by pid in the structure QP0W_PID_Entry_T.

## Parameters

***mbrinfo***

(I/O) A pointer to the QP0W_PID_Entry_T structure. This structure contains the process table entry information for the processes that are members of the process group identified by pid.

The structure QP0W_PID_Entry_T is defined in the **<qp0wpid.h>** header file as follows:

```
typedef struct QP0W_PID_Entries_T {
    int           entries_prov;
    int           entries_could;
    int           entries_return;
    pid_t         pid;
    QP0W_PID_Data_T  entry[1];
} QP0W_PID_Entries_T;
```

The members of the QP0W_PID_Entry_T structure are as follows:

| | |
|---|---|
| *int entries_prov;* | (Input) The number of entries of type QP0W_PID_Data_T for which the caller has allocated storage to contain the status and process table entry information. |
| *int entries_could;* | (Output) The number of entries of type QP0W_PID_Data_T that could be returned. If the entries_could value exceeds the entries_prov value, the **Qp0wChkPgrp()** function should be called again with sufficient storage to contain the number of entries returned in entries_could (entries_prov must be greater than or equal to entries_could). |

| | |
|---|---|
| *int entries_return;* | (Output) The number of entries of type QP0W_PID_Data_T that were returned. If the entries_return value is less than the entries_prov value, the content of the excess number of entries provided is unchanged by **Qp0wChkPgrp()**. |
| *pid_t pid;* | (Input) The process group ID of the group of processes for which the process information is to be returned. |
| *QP0W_PID_Data_T entry[1];* | (Output) The process table information for the process group members. There is one QP0W_PID_Data_T structure entry for each process group member, limited by the value of entries_prov. |

The structure QP0W_PID_Data_T is defined in the **<qp0wpid.h>** file as follows:

```
typedef struct QP0W_PID_Data_T {
    pid_t          pid;
    pid_t          ppid;
    pid_t          pgrp;
    int            status;
    unsigned int  exit_status;
} QP0W_PID_Data_T;
```

The members of the QP0W_PID_Data_T structure are as follows:

| | |
|---|---|
| *pid_t pid;* | The process ID of the process. |
| *pid_t ppid;* | The process ID of the parent process. If ppid has a value of binary 1, there is no parent process associated with the process. |
| *pid_t pgrp;* | The process group ID of the process. |
| *int status;* | A collection of flag bits that describe the current state of the process. The following flag bits can be set in status: |

| | |
|---|---|
| *QP0W_PID_TERMINATED* | The process has ended. |
| *QP0W_PID_STOPPED* | The process was stopped by a signal. |
| *QP0W_PID_CHILDWAIT* | The process is waiting for a child process to be ended or stopped by a signal. |
| *QP0W_PID_SIGNALSTOP* | The process has requested that the SIGCHLD signal be generated for the process when one of its child processes is stopped by a signal. |

| | |
|---|---|
| *unsigned int exit_status;* | Exit status of the process. This member only has meaning if the status is set to QP0W_PID_TERMINATED. Refer to the **wait()** function for a description of the exit status for a process. |

## Authorities

The process calling **Qp0wChkPgrp()** must have the appropriate authority to the processes being examined. A process is allowed to examine the process table information for a process if at least one of the following conditions is true:

- The process is calling **Qp0wChkPgrp()** for its own process.
- The process has *JOBCTL special authority defined in the process user profile or in a current adopted user profile.
- The process is the parent of the process (the process being examined has a parent process ID equal to the process ID of the process calling **Qp0wChkPgrp**()).
- The real or effective user ID of the process matches the real or effective user ID of the process calling **Qp0wChkPgrp**().

## Return Value

*0* **Qp0wChkPgrp()** was successful.

*value* **Qp0wChkPgrp()** was not successful. The value returned indicates one of the following errors. Under some conditions, *value* could indicate an error other than those listed here.

*[EINVAL]* An invalid parameter was found.

A parameter passed to this function is not valid.

*[EPERM]* Operation not permitted.

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

*[ESRCH]* No item could be found that matches the specified value.

## Usage Notes

The **Qp0wChkPgrp()** function provides an OS/400-specific way to obtain the process table information for the members of a process group.

## Related Information

- The <**qp0wpid.h**> file (see Header Files for UNIX-Type Functions)
- The <**signal.h**> file (see Header Files for UNIX-Type Functions)

- getpgrp()--Get Process Group ID
- getpid()--Get Process ID
- getppid()--Get Process ID of Parent Process
- Qp0wGetPgrp()--Get Process Group ID
- Qp0wGetPid()--Get Process ID
- Qp0wGetPPid()--Get Process ID of Parent Process
- wait()--Wait for Child Process to End

---

API introduced: V3R6

---

# Qp0wChkPid()--Check Status for Process ID

Syntax

```
#include <sys/types.h>
#include <qp0wpid.h>

int Qp0wChkPid(pid_t pid,
               QP0W_PID_Data_T *pidinfo);
```

Service Program Name: QP0WPID

Default Public Authority: *USE

Threadsafe: Yes

The **Qp0wChkPid()** function returns the status and process table entry information for the process specified by the process ID pid.

## Parameters

***pid***

(Input) The process ID of the process whose process table information is to be returned. When pid has a value of binary 0, the process table information for the current process is returned.

***pidinfo***

(Output) A pointer to the QP0W_PID_Data_T structure. The process table entry information for the process identified by pid is stored in the location pointed to by the pidinfo parameter.

The structure QP0W_PID_Data_T is defined in **<qp0wpid.h>** header file as follows:

```
typedef struct QP0W_PID_Data_T {
    pid_t          pid;
    pid_t          ppid;
    pid_t          pgrp;
    int            status;
    unsigned int   exit_status;
} QP0W_PID_Data_T;
```

The members of the QP0W_PID_Data_T structure are as follows:

| | |
|---|---|
| *pid_t pid;* | The process ID of the process. |
| *pid_t ppid;* | The process ID of the parent process. If ppid has a value of binary 1, there is no parent process associated with the process. |
| *pid_t pgrp;* | The process group ID of the process. |

| | |
|---|---|
| *int status;* | A collection of flag bits that describe the current state of the process. The following flag bits can be set in status: |

| | |
|---|---|
| *QP0W_PID_TERMINATED* | The process has ended. |
| *QP0W_PID_STOPPED* | The process has been stopped by a signal. |
| *QP0W_PID_CHILDWAIT* | The process is waiting for a child process to be ended or stopped by a signal. |
| *QP0W_PID_SIGNALSTOP* | The process has requested that the SIGCHLD signal be generated for the process when one of it's child processes has been stopped by a signal. |

| | |
|---|---|
| *unsigned int exit_status;* | Exit status of the process. This member only has meaning if the status has been set to QP0W_PID_TERMINATED. Refer to the **wait**() function for a description of the exit status for a process. |

## Authorities

The process calling **Qp0wChkPid()** must have the appropriate authority to the process being examined. A process is allowed to examine the process table information for a process if at least one of the following conditions is true:

- The process is calling **Qp0wChkPid()** for its own process.
- The process has *JOBCTL special authority defined in the process user profile or in a current adopted user profile.
- The process is the parent of the process (the process being examined has a parent process ID equal to the process ID of the process calling **Qp0wChkPid**()).
- The real or effective user ID of the process matches the real or effective user ID of the process calling **Qp0wChkPid**().

## Return Value

| | |
|---|---|
| *0* | **Qp0wChkPid**() was successful. |

*value* **Qp0wChkPid()** was not successful. The value returned indicates one of the following errors. Under some conditions, *value* could indicate an error other than those listed here.

*[EINVAL]* An invalid parameter was found.

A parameter passed to this function is not valid.

*[EPERM]* Operation not permitted.

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

*[ESRCH]* No item could be found that matches the specified value.

## Usage Notes

The **Qp0wChkPid()** function provides an OS/400-specific way to obtain the process table information for the specified process.

## Related Information

- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- The <**qp0wpid.h**> file (see Header Files for UNIX-Type Functions)
- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- getpgrp()--Get Process Group ID
- getpid()--Get Process ID
- getppid()--Get Process ID of Parent Process
- Qp0wGetPgrp()--Get Process Group ID
- Qp0wGetPid()--Get Process ID
- Qp0wGetPPid()--Get Process ID of Parent Process
- wait()--Wait for Child Process to End

---

API introduced: V3R6

---

# Qp0wGetJobID()--Get Qualified Job Name and ID for Process ID

Syntax

```
#include <qp0wpid.h>

int Qp0wGetJobID(pid_t pid, QP0W_Job_ID_T *jobinfo);
```

Service Program Name: QP0WPID

Default Public Authority: *USE

Threadsafe: Yes

The **Qp0wGetJobID()** function returns the qualified job name and internal job identifier for the process whose process ID matches pid.

## Parameters

***pid***

(Input) The process ID of the process whose job number is to be returned. When pid has a value of zero, the process ID of the calling process is used.

***jobinfo***

(Output) A pointer to the `QP0W_Job_ID_T` structure. This structure contains the qualified OS/400 job name and internal job identifier for the process identified by pid.

The structure `QP0W_Job_ID_T` is defined in the **<qp0wpid.h>** header file as follows:

```
typedef struct QP0W_Job_ID_T {
    char    jobname[10];
    char    username[10];
    char    jobnumber[6];
    char    jobid[16];
} QP0W_Job_ID_T;
```

The members of the `QP0W_Job_ID_T` structure are as follows:

| | |
|---|---|
| *char jobname[10]* | The name of the job as identified to the system. For an interactive job, the system assigns the job the name of the work station where the job started. For a batch job, you specify the name in the command when you submit the job. |
| *char username[10]* | The user name under which the job runs. The user name is the same as the user profile name and can come from several different sources, depending on the type of job. |
| *char jobnumber[6]* | The system-generated job number. |

*char jobid[16]* The internal job identifier. This value is sent to other APIs to speed the process of locating the job on the system. The identifier is not valid following an initial program load (IPL). If you attempt to use it after an IPL, an exception occurs.

## Authorities

The process calling **Qp0wGetJobID()** must have the appropriate authority to the process whose job number is to be returned. A process is allowed to access the job number for a process if at least one of the following conditions is true:

- The process is calling **Qp0wGetJobID()** for its own process.
- The process has *JOBCTL special authority defined in the process user profile or in a current adopted user profile.
- The process is the parent of the process (the process being examined has a parent process ID equal to the process ID of the process calling **Qp0wGetJobID()**).
- The real or effective user ID of the process matches the real or effective user ID of the process calling **Qp0wGetJobID()**.

## Return Value

*0* **Qp0wGetJobID()** was successful.

*value* **Qp0wGetJobID()** was not successful. The value returned indicates one of the following errors. Under some conditions, *value* could indicate an error other than those listed here.

*[EINVAL]* An invalid parameter was found.

A parameter passed to this function is not valid.

*[EPERM]* Operation not permitted.

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

*[ESRCH]* No item could be found that matches the specified value.

## Related Information

- The <**qp0wpid.h**> file (see Header Files for UNIX-Type Functions)
- getpid()--Get Process ID

- Qp0wGetPid()--Get Process ID

---

API introduced: V3R6

---

Top | UNIX-Type APIs | APIs by category

# Qp0wGetPgrp()--Get Process Group ID

Syntax

```
#include <sys/types.h>
#include <qp0wpid.h>

pid_t Qp0wGetPgrp(void);
```

Service Program Name: QP0WSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **Qp0wGetPgrp()** function returns the process group ID of the calling process.

## Parameters

None.

## Authorities

None.

## Return Value

| | |
|---|---|
| *pid_t* | The value returned by **Qp0wGetPgrp()** is the process group ID of the calling process. |

## Error Conditions

The **Qp0wGetPgrp()** function is always successful and does not return an error.

## Usage Notes

1. The **Qp0wGetPgrp()** function provides an OS/400-specific way to obtain the process group ID of the calling process. It performs the same function as **getpgrp()**.

2. **Qp0wGetPgrp()** enables a process for signals if the process is not already enabled for signals. For details, see Qp0sEnableSignals()--Enable Process for Signals.

## Related Information

- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- getpgrp()--Get Process Group ID
- Qp0sDisableSignals()--Disable Process for Signals
- Qp0sEnableSignals()--Enable Process for Signals

---

API introduced: V3R6

---

# Qp0wGetPid()--Get Process ID

Syntax

```
#include <sys/types.h>
#include <qp0wpid.h>

pid_t Qp0wGetPid(void);
```

Service Program Name: QP0WSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **Qp0wGetPid()** function returns the process ID of the calling process.

## Parameters

None.

## Authorities

None.

## Return Value

| | |
|---|---|
| *pid_t* | The value returned by **Qp0wGetPid()** is the process ID of the calling process. |

## Error Conditions

The **Qp0wGetPid()** function is always successful and does not return an error.

## Usage Notes

1. The **Qp0wGetPid()** function provides an OS/400-specific way to obtain the process ID of the calling process. It performs the same function as **getpid()**.

2. **Qp0wGetPid()** enables a process for signals if the process is not already enabled for signals. For details, see (see Qp0sEnableSignals()--Enable Process for Signals.

## Related Information

- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- The <**qp0wpid.h**> file (see Header Files for UNIX-Type Functions)
- getpid()--Get Process ID
- Qp0sDisableSignals()--Disable Process for Signals
- Qp0sEnableSignals()--Enable Process for Signals
- Qp0wGetPidNoInit()--Get Process ID without Initializing for Signals

---

API introduced: V3R6

---

# Qp0wGetPidNoInit()--Get Process ID without Initializing for Signals

Syntax

```
 #include <sys/types.h>
 #include <qp0wpid.h>

 pid_t Qp0wGetPidNoInit(void);
```

Service Program Name: QP0WSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **Qp0wGetPidNoInit()** function returns the process ID of the calling process without enabling the process to receive signals.

## Parameters

None.

## Authorities

None.

## Return Value

*pid_t* The value returned by **Qp0wGetPidNoInit()** is the process ID of the calling process.

## Error Conditions

The **Qp0wGetPidNoInit()** function is always successful and does not return an error.

## Usage Notes

The **Qp0wGetPidNoInit()** function provides an OS/400-specific way to obtain the process ID of the calling process. It performs the same function as the **getpid()** function without enabling the process to receive signals.

## Related Information

- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- The <**qp0wpid.h**> file (see Header Files for UNIX-Type Functions)
- getpid()--Get Process ID
- Qp0wGetPid()--Get Process ID
- Qp0sDisableSignals()--Disable Process for Signals
- Qp0sEnableSignals()--Enable Process for Signals

---

API introduced: V3R6

---

# Qp0wGetPPid()--Get Process ID of Parent Process

Syntax

```
 #include <sys/types.h>
 #include <qp0wpid.h>

 pid_t Qp0wGetPPid(void);
```

Service Program Name: QP0WSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **Qp0wGetPPid()** function returns the parent process ID of the calling process.

## Parameters

None.

## Authorities

None.

## Return Value

*pid_t* The value returned by **Qp0wGetPPid()** is the process ID of the parent process for the calling process. A process ID value of 1 indicates that there is no parent process associated with the calling process.

## Error Conditions

The **Qp0wGetPPid()** function is always successful and does not return an error.

## Usage Notes

The **Qp0wGetPPid()** function provides an OS/400-specific way to obtain the parent process ID of the calling process. It performs the same function as **getppid()**.

## Related Information

- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- The <**qp0wpid.h**> file (see Header Files for UNIX-Type Functions)
- getppid()--Get Process ID of Parent Process

---

API introduced: V3R6

---

# 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

---

# Qp0zSystem()--Run a CL Command

Syntax

```
#include <qp0z1170.h>

int Qp0zSystem( const char *CLcommand );
```

Service Program Name: QP0ZTRML

Default Public Authority: *USE

Threadsafe: Yes

The **Qp0zSystem()** function spawns a new process, passes *CLcommand* to the CL command processor in the new process, and waits for the command to complete. The command runs in a batch job so it does not have access to a terminal.

This function is similar to the **system()** function provided by ILE C, but allows a program to safely run a CL command from a multithreaded process. Note that if *CLcommand* fails, the global variable _EXCP_MSGID is not set with the exception message id.

## Parameters

***CLcommand***
: (Input) Pointer to null-terminated CL command string.

## Authorities

The user calling **Qp0zSystem()** must have *USE authority to the specified CL command.

## Return Value

| | |
|---|---|
| *0* | The specified CL command was successful. |
| *1* | The specified CL command was not successful. |
| *-1* | **Qp0zSystem()** was not successful. |

## Related Information

- The <**qp0z1170.h**> file (see Header Files for UNIX-Type Functions)

## Example

See Code disclaimer information for information pertaining to code examples.

The following example shows how to use the **Qp0zSystem()** function to create a library.

```
#include <stdio.h>
#include <qp0z1170.h>

int main(int argc, char *argv[])
{
  if (Qp0zSystem("CRTLIB LIB(XYZ)") != 0)
    printf("Error creating library XYZ.\n");
  else
    printf("Library XYZ created.\n");

  return(0);
}
```

## Output:

```
    Library XYZ created
```

---

API introduced: V4R2

---

# setpgid()--Set Process Group ID for Job Control

Syntax

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

int setpgid(pid_t pid, pid_t pgid);
```

Service Program Name: QP0WSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **setpgid()** function is used to either join an existing process group or create a new process group within the session of the calling process.

See the Usage Notes for considerations in using **setpgid()**.

## Parameters

***pid***

(Input) The process ID of the process whose process group ID is to be changed. When pid has a value of zero, the process group ID of the calling process is changed.

***pgid***

(Input) The process group ID to be assigned to the process whose process ID matches pid. The value of pgid must be within the range of zero through the maximum signed integer. When pgid has a value of zero, the process group ID is set to the process ID of the process indicated by pid.

## Authorities

The process calling **setpgid()** must have the appropriate authority to the process being changed. A process is allowed to access the process group ID for a process if at least one of the following conditions is true:

- The process is calling **setpgid()** for its own process.
- The process has *JOBCTL special authority defined in the process user profile or in a current adopted user profile.
- The process is the parent of the process (the process being examined has a parent process ID equal to the process ID of the process calling **setpgid()**).

- The real or effective user ID of the process matches the real or effective user ID of the process calling **setpgid**()).

## Return Value

*0* **setpgid()** was successful.

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

## Error Conditions

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

*[EINVAL]* An invalid parameter was found.

A parameter passed to this function is not valid.

*[EPERM]* Operation not permitted.

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

*[ESRCH]* No item could be found that matches the specified value.

## Usage Notes

1. OS/400 does not support sessions. Until session support is available on OS/400, the restriction that the process group must be within the session of the calling process will not be enforced.

2. The **setpgid()** function fails if a nonzero process group ID is specified and that process group does not exist. If this occurs, the return value is set to -1 and errno is set to [EPERM].

## Related Information

- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- getpgrp()--Get Process Group ID
- Qp0wGetPgrp()--Get Process Group ID

API introduced: V3R6

# »setrlimit()--Set resource limit

Syntax

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

int setrlimit(int resource, const struct rlimit *rlp);
```

Service Program Name: QP0WSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **setrlimit()** function sets the resource limit for the specified *resource*. A resource limit is a way for the operating system to enforce a limit on a variety of resources used by a process. A resource limit is represented by a rlimit structure. The rlim_cur member specifies the current or soft limit and the rlim_max member specifies the maximum or hard limit.

A soft limit can be changed to any value that is less than or equal to the hard limit. The hard limit can be changed to any value that is greater than or equal to the soft limit. Only a process with appropriate authorities can increase a hard limit.

The **setrlimit()** function supports the following resources:

*RLIMIT_FSIZE (0)* The maximum size of a file in bytes that can be created by a process.

The **setrlimit()** function does not support setting the following resources: RLIMIT_AS, RLIMIT_CORE, RLIMIT_CPU, RLIMIT_DATA, RLIMIT_NOFILE, and RLIMIT_STACK. The **setrlimit()** function returns -1 and sets errno to ENOTSUP when called with one of these resources.

The value of RLIM_INFINITY is considered to be larger than any other limit value. If the value of the limit is set to RLIM_INFINITY, then a limit is not enforced for that resource. If the value of the limit is set to RLIM_SAVED_MAX, the new limit is the corresponding saved hard limit. If the value of the limit is RLIM_SAVED_CUR, the new limit is the corresponding saved soft limit.

## Parameters

***resource***

(Input)

The resource to set the limits for.

***rlp**

(Input)

Pointer to a struct rlim_t that contains the new values for the hard and soft limits.

## Authorities and Locks

The current user profile must have *JOBCTL special authority to increase the hard limit.

## Return Value

**0** **setrlimit**() was successful.

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

## Error Conditions

If **setrlimit**() 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]
: An invalid parameter was found.

  An invalid *resource* was specified.

  The new soft limit is greater the new hard limit.

  The new hard limit is lower than the new soft limit.

[EPERM]
: Permission denied.

  An attempt was made to increase the hard limit and the current user profile does not have *JOBCTL special authority.

[ENOTSUP]
: Operation not supported.

  The operation, though supported in general, is not supported for the requested *resource*.

## Related Information

- The <**sys/resource.h**> file (see Header Files for UNIX-Type Functions)
- getrlimit()-Get resource limit

- ulimit()-Get and set process limits

## Example

```
#include <sys/resource.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main (int argc, char *argv[])
{
  struct rlimit limit;

  /* Set the file size resource limit. */
  limit.rlim_cur = 65535;
  limit.rlim_max = 65535;
  if (setrlimit(RLIMIT_FSIZE, &limit) != 0) {
    printf("setrlimit() failed with errno=%d\n", errno);
    exit(1);
  }

  /* Get the file size resource limit. */
  if (getrlimit(RLIMIT_FSIZE, &limit) != 0) {
    printf("getrlimit() failed with errno=%d\n", errno);
    exit(1);
  }

  printf("The soft limit is %llu\n", limit.rlim_cur);
  printf("The hard limit is %llu\n", limit.rlim_max);
  exit(0);
}
```

**Example Output:**

```
The soft limit is 65535
The hard limit is 65535
```

«

---

Introduced: V5R2

---

# spawn()--Spawn Process

Syntax

```
#include <spawn.h>

pid_t spawn( const char                    *path,
             const int                     fd_count,
             const int                     fd_map[],
             const struct inheritance      *inherit,
             char * const                  argv[],
             char * const                  envp[]);
```

Service Program Name: QP0ZSPWN

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **spawn()** function creates a child process that inherits specific attributes from the parent. The attributes inherited by the child process are file descriptors, the signal mask, the signal action vector, and environment variables, among others.

## Parameters

***path***

(Input) Specific path to an executable file that will run in the new (child) process. The path name is expected to be in the CCSID of the job.

See QlgSpawn()--Spawn Process (using NLS-enabled path name) for a description and an example of supplying the *path* in any CCSID.

***fd_count***

(Input) The number of file descriptors the child process can inherit. It can have a value from zero to the value returned from a call to **sysconf(**_SC_OPEN_MAX).

***fd_map[]***

(Input) An array that maps the parent process file descriptor numbers to the child process file descriptor numbers. If this value is NULL, it indicates simple inheritance. **Simple inheritance** means that the child process inherits all eligible open file descriptors of the parent process. In addition, the file descriptor number in the child process is the same as the file descriptor number in the parent process. Refer to Attributes Inherited for details of file descriptor inheritance.

***inherit***

(Input) A pointer to an area of type struct inheritance. If the pointer is NULL, an error occurs. The

inheritance structure contains control information to indicate attributes the child process should inherit from the parent. The following is an example of the inheritance structure, as defined in the **<spawn.h>** header file:

```
 struct inheritance {
                    flagset_t  flags;
                    int        pgroup;
                    sigset_t   sigmask;
                    sigset_t   sigdefault;
};
```

The flags field specifies the manner in which the child process should be created. Only the constants defined in **<spawn.h>** are allowed; otherwise, *spawn* returns -1 with errno set to EINVAL. The allowed constants follow:

*SPAWN_SETPGROUP*

If this flag is set ON, **spawn()** sets the process group ID of the child process to the value in pgroup. In this case, the process group field, pgroup, must be valid. If it is not valid, an error occurs. If this flag is set OFF, the pgroup field is checked to determine what the process group ID of the child process is set to. If the pgroup field is set to the constant SPAWN_NEWPGROUP, the child process group ID is set to the child process ID. If the pgroup field is not set to SPAWN_NEWPGROUP and the flags field is not set to SPAWN_SETPGROUP, the process group ID of the child process is set to the process group ID of the parent process. If the pgroup field is set to SPAWN_NEWPGROUP and the flags field is set to SPAWN_SETPGROUP, an error occurs.

*SPAWN_SETSIGMASK*

If this flag is set ON, **spawn()** sets the signal blocking mask of the child process to the value in *sigmask*. In this case, the signal blocking mask must be valid. If it is not valid, an error occurs. If this flag is set OFF, **spawn()** sets the signal blocking mask of the child process to the signal blocking mask of the calling thread.

*SPAWN_SETSIGDEF*

If this flag is set ON, **spawn()** sets the child process' signals identified in sigdefault to the default actions. The sigdefault must be valid. If it is not valid, an error occurs. If this flag is set OFF, **spawn()** sets the child process' signal actions to those of the parent process. Any signals of the parent process that have a catcher specified are set to default in the child process. The child process' signal actions inherit the parent process' ignore and default signal actions.

*SPAWN_SETTHREAD_NP*

If this flag is set ON, **spawn()** will create the child process as multithread capable. The child process will be allowed to create threads. If this flag is set OFF, the child process will not be allowed to create threads.

**Note:** The SPAWN_SETTHREAD_NP flag is a non-standard, OS/400-platform-specific extension to the inheritance structure. Applications that wish to avoid using platform-specific extensions should not use this flag.

*SPAWN_SETPJ_NP*

If this flag is set ON, **spawn()** attempts to use available OS/400 prestart jobs. The prestart job entries that may be used follow:

- QSYS/QP0ZSPWP, if the flag SPAWN_SETTHREAD_NP is set OFF.
- QSYS/QP0ZSPWT, if the flag SPAWN_SETTHREAD_NP is set ON.

The OS/400 prestart jobs must have been started using either QSYS/QP0ZSPWP or QSYS/QP0ZSPWT as the program that identifies a prestart job entry for the OS/400 subsystem that the parent process is running under. If a prestart job entry is not defined, the child process will run as a batch immediate job under the same subsystem as the parent process.

If this flag (SPAWN_SETPJ_NP) is set OFF, the child process will run as a batch immediate job under the same subsystem as the parent process.

**Notes:**

1. In order to more closely emulate POSIX semantics, **spawn()** will ignore the Maximum number of uses (MAXUSE) value specified for the prestart job entry. The prestart job will only be used once, behaving as if MAXUSE(1) was specified.
2. The SPAWN_SETPJ_NP flag is a non-standard, OS/400-platform-specific extension to the inheritance structure. Applications that wish to avoid using platform-specific extensions should not use this flag.

| | |
|---|---|
| *SPAWN_SETCOMPMSG_NP* | If this flag is set ON, **spawn()** causes the child process to send a completion message to the user's message queue when the child process ends. If this flag is set OFF, no completion message is sent to the user's message queue when the child process ends. If both the SPAWN_SETCOMPMSG_NP and SPAWN_SETPJ_NP flags are set ON, an error occurs.<br><br>**Note:**The SPAWN_SETCOMPMSG_NP flag is a non-standard, OS/400-platform-specific extension to the inheritance structure. Applications that wish to avoid using platform-specific extensions should not use this flag. |
| *SPAWN_SETJOBNAMEPARENT_NP* | If this flag is set ON, **spawn()** set the child's OS/400 simple job name to that of the parent's. If this flag is set OFF, **spawn()** sets the child's OS/400 simple job name based on the path input parameter. |

***argv[]***

(Input) An array of pointers to strings that contain the argument list for the executable file. The last element in the array must be the NULL pointer. If this parameter is NULL, an error occurs.

***envp[]***

(Input) An array of pointers to strings that contain the environment variable lists for the executable file. The last element in the array must be the NULL pointer. If this parameter is NULL, an error occurs.

## Authorities

**Figure 1-3. Authorization Required for spawn()**

| Object Referred to | Authority Required | errno |
|---|---|---|
| Each directory in the path name preceding the executable file that will run in the new process | *X | EACCES |
| Executable file that will run in the new process | *X | EACCES |
| If executable file that will run in the new process is a shell script | *RX | EACCES |

## Return Value

| | |
|---|---|
| *value* | **spawn()** was successful. The value returned is the process ID of the child process. |
| *-1* | **spawn()** was not successful. The *errno* variable is set to indicate the error. |

## Error Conditions

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

*[E2BIG]*
: Argument list too long.

*[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.

*[EAPAR]*
: Possible APAR condition or hardware failure.

*[EBADFUNC]*
: Function parameter in the signal function is not set.

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

*[EBADNAME]*
: The object name specified is not correct.

*[ECANCEL]*
: Operation canceled.

*[ECONVERT]*
: Conversion error.

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

  The specified path name is not in the CCSID of the job.

*[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.

The flags field in the inherit parameter contains an invalid value.

*[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.

*[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.

*[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.

*[ETERM]* Operation terminated.

*[ENOSYSRSC]* System resources not available to complete request.

The child process failed to start. The maximum active jobs in a subsystem may have been reached. CHGSBSD and CHGJOBQE CL commands can be used to change the maximum active jobs.

*[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

1. **spawn()** is threadsafe, except this function will fail and errno ENOTSAFE will be set if it is called in any of the following ways:

   - From a multithreaded process and path refers to a shell script that does not exist in a threadsafe file system.

2. There are performance considerations when using **spawn()** and **spawnp()** concurrently among threads in the same process. **spawn()** and **spawnp()** serialize against other **spawn()** and **spawnp()** calls from other threads in the same process.

3. The child process is enabled for signals. A side effect of this function is that the parent process is also enabled for signals if it was not enabled for signals before this function was called.

4. If this function is called from a program running in user state and it specifies a system-domain program as the executable program for the child process, an exception occurs. In this case, **spawn()** returns the process ID of the child process. On a subsequent call to **wait()** or **waitpid()**, the status information returned indicates that an exception occurred in the child process.

5. The program that will be run in the child process must be either a program object in the QSYS.LIB file system or an independent ASP QSYS.LIB file system (*PGM object) or a shell script (see About Shell Scripts). The syntax of the name of the file to run must be the proper syntax for the file system in which the file resides. For example, if the program MYPROG resides in the QSYS.LIB file system and in library MYLIB, the specification for **spawn()**. would be the following:

   ```
   /QSYS.LIB/MYLIB.LIB/MYPROG.PGM
   ```

   See QlgSpawn()--Spawn Process (using NLS-enabled path name) for an example specifying the program using the Qlg_Path_Name_T structure. The Qlg_Path_Name_T structure is supported by

**QlgSpawn()** and allows the program name to be specified in any CCSID.

**Note:** For more information about path syntaxes for the different file systems, see the Integrated File System book.

6. Spawned child processes are batch jobs or prestart jobs. As such, they do not have the ability to do 5250-type interactive I/O.

7. Spawned child processes that are OS/400 prestart jobs are similar to batch jobs. Due to the nature of prestart jobs, only the following OS/400-specific attributes are explicitly inherited in a child process when you use prestart jobs:

   - Library list
   - Language identifier
   - Country or region identifier
   - Coded character set identifier
   - Default coded character set identifier
   - Locale (as specified in the user profile)

   The child process has the same user profile as the calling thread. However, the OS/400 job attributes come from the job description specified for the prestart job entry, and the run attributes come from the class that is associated with the OS/400 subsystem used for the prestart job entry.

   **Notes:**

   1. The prestart job entry QP0ZSPWP is used with prestart jobs that will not be creating threads. The prestart job entry QP0ZSPWT is used with prestart jobs that will allow multiple threads. Both types of prestart jobs may be used in the same subsystem. The prestart job entry must be defined for the subsystem that the **spawn()** parent process runs under in order for it to be used.

   2. The following example defines a prestart job entry (QP0ZSPWP) for use by **spawn()** under the subsystem QINTER. The **spawn()** API must have the SPAWN_SETPJ_NP flag set (but not SPAWN_SETTHREAD_NP) in order to use these prestart jobs:

      ```
      ADDPJE SBSD(QSYS/QINTER) PGM(QSYS/QP0ZSPWP)
             INLJOBS(20) THRESHOLD(5) ADLJOBS(5)
             JOBD(QGPL/QDFTJOBD) MAXUSE(1)
             CLS(QGPL/QINTER)
      ```

   3. The following example defines a prestart job entry (QP0ZSPWT) that will create prestart jobs that are multithread capable for use by **spawn()** under the subsystem QINTER. The **spawn()** API must have both SPAWN_SETPJ_NP and SPAWN_SETTHREAD_NP flags set in order to use these prestart jobs. Also, the JOBD parameter must be a job description that allows multiple threads as follows:

      ```
      ADDPJE SBSD(QSYS/QINTER) PGM(QSYS/QP0ZSPWT)
             INLJOBS(20) THRESHOLD(5) ADLJOBS(5)
             JOBD(QSYS/QAMTJOBD) MAXUSE(1)
             CLS(QGPL/QINTER)
      ```

   Refer to the Work Management book on the V5R1 Supplemental Manuals Web site for complete details on prestart jobs.

8. Shell scripts are allowed for the child process. If a shell script is specified, the appropriate shell interpreter program is called. The shell script must be a text file and must contain the following format on the first line of the file:

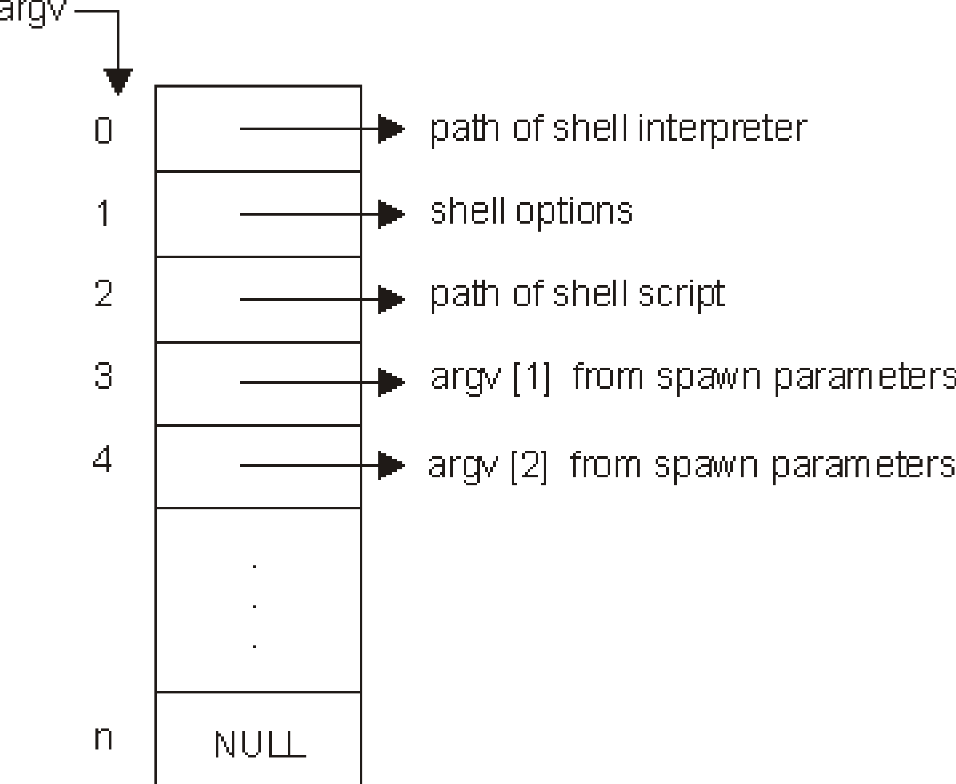
   ```
   #!interpreter_path <options>
   ```

   where interpreter_path is the path to the shell interpreter program.

   If the calling process is multithreaded, path (the first parameter to **spawn()**) must reference a threadsafe file system.

   **spawn()** calls the shell interpreter, passing in the shell options and the arguments passed in as a parameter to **spawn()**. The argument list passed into the shell interpreter will look like Figure 1-4.

   **Figure 1-4. Arguments to Shell Interpreter**

   See About Shell Scripts for an example using **spawn()** and shell scripts.

9. Only programs that expect arguments as NULL-terminated strings can be spawned.

   The program that is run in the child's process is called at its initial entry point. The linkage to the program is C-like. The following example describes the linkage in C language terms.

   ```
   int main(int argc, char *argv[])
   [
   ]
   ```

   where there following are true:

   - *argc* is the number of arguments in *argv[]*.

- *argv[]* is an array of arguments represented as strings. The last entry in the array is NULL. The first entry in the array, by convention, is the name of the program. **spawn()** sets the element argv[0] to the path name of the child process' program. **spawn()** does not move any elements of the *argv* array when it sets argv[0] to the path name of the child process' program. If that element of the array contains an argument value, the value is overwritten.

*argv[]* is specified by the user on the interface to **spawn()**. When **spawn()** is called in the child's process, it passes the array to the program.

10. The child process does not inherit any of the environment variables of the parent process. That is, the default environment variable environment is empty. If the child process is to inherit all the parent process' environment variables, the extern variable *environ* can be used as the value for *envp[]* when **spawn()** is called. If a specific set of environment variables is required in the child process, the user must build the *envp[]* array with the "name=value" strings. In the child process, **spawn()** does the equivalent of a **putenv** on each element of the *envp[]* array. Then the extern variable *environ* will be set and available to the child process' program.

    **Note:** If the user of **spawn()** specifies the extern variable *environ* as the *envp[]* parameter, the user must successfully call one of the following APIs before calling **spawn()**:

    - getenv()
    - putenv()
    - Qp0zGetEnv()
    - Qp0zInitEnv()
    - Qp0zPutEnv()

    The extern variable *environ* is not initialized until one of these APIs is called in the current activation group. If *environ* is used in a call to **spawn()** without first calling one of these APIs, **spawn()** returns an error.

11. OS/400 handles stdin, stdout, and stderr differently than most UNIX systems. On most UNIX systems, stdin, stdout, and stderr have file descriptors 0, 1, and 2 reserved and allocated for them. On OS/400, this is not the case. There are two ramifications of this difference:

    1. File descriptor 0, 1, and 2 are allocated to the first three files that have descriptors allocated to them. If an application writes to file descriptor 1 assuming it is stdout, the result will not be as expected.

    2. Any API that assumes stdin, stdout, and stderr are file descriptors 0, 1, and 2 will not behave as expected.

    Users and applications can enable descriptor-based standard I/O for child processes by setting environment variable QIBM_USE_DESCRIPTOR_STDIO to the value Y in the child process. This can be accomplished on the call to **spawn()** by either of the following:

    1. Specifying the extern variable *environ* as the *envp[]* parameter. This assumes that the QIBM_USE_DESCRIPTOR_STDIO environment variable exists in the calling process.

        The environment variable can be set by using one of the following:

        - API putenv("QIBM_USE_DESCRIPTOR_STDIO=Y");
        - Command ADDENVVAR ENVVAR(QIBM_USE_DESCRIPTOR_STDIO) VALUE(Y)
        - Command CHGENVVAR ENVVAR(QIBM_USE_DESCRIPTOR_STDIO) VALUE(Y)

2. Explicitly include "QIBM_USE_DESCRIPTOR_STDIO=Y" in the user-defined *envp[]* array with the "name=value" strings.

If you enable descriptor-based standard I/O for child processes, file descriptors 0, 1, and 2 are automatically used for stdin, stdout, and stderr, respectively. However, **spawn()** must be called using a *fd_map* that has file descriptors 0, 1, and 2 properly allocated. See About Shell Scripts for an example that enables descriptor-based standard I/O for a child process. Refer to the WebSphere Development Studio: ILE C/C++ Programmer's Guide for complete details on this support.

12. Spawn users have a facility to aid in debugging child processes.

    To help the user start a debug session (when **spawn()** is the mechanism used to start the process), the user sets the environment variable QIBM_CHILD_JOB_SNDINQMSG.

    If the environment variable is assigned a numerical value, it indicates the number of descendent levels that will be enabled for debugging. This support can be used to debug applications that create children, grandchildren, great-grandchildren, and so forth. When the environment variable has a value of 1, it enables debugging of all subsequent child processes. A value of 2 enables debugging of all subsequent child processes and grandchild processes.

    When the environment variable has a value less than or equal to 0, or any non-numerical value, debugging will not occur.

    Here are the steps a user would take to debug an application by using **spawn()**:

    Assume the user wants to debug child processes in an application called CHILDAPP found in library MYAPPLIB.

    - Set the QIBM_CHILD_JOB_SNDINQMSG environment variable to 1.

      The environment variable can be set by using one of the following:

      - API putenv("QIBM_CHILD_JOB_SNDINQMSG=1");
      - Command ADDENVVAR ENVVAR(QIBM_CHILD_JOB_SNDINQMSG) VALUE(1)
      - Command CHGENVVAR ENVVAR(QIBM_CHILD_JOB_SNDINQMSG) VALUE(1)

    - Call or run the application that specifies /QSYS.LIB/MYAPPLIB.LIB/CHILDAPP.PGM as the pathon the **spawn()** invocation. CHILDAPP will start running, send a CPAA980 *INQUIRY message to the user's message queue, and then will block, waiting for a reply to the message. Issue a Work with Active Jobs (WRKACTJOB) command and find the CHILDAPP in a MSGW job status. Option 7 (Display message) performed against this job will display the CPAA980 *INQUIRY message that was sent. As part of this message, the Qualified Job Name will be displayed in the proper format to pass to the Start Service Job (STRSRVJOB) command (for example, 145778/RANDYR/CHILDAPP).

      **Note:** Alternatively, a Display Messages (DSPMSG) command can be issued for the user, and the output searched for the specific CPAA980 *INQUIRY message.

      **Note:** If the job's inquiry message reply specifies using the default message reply, the child process will not block since the default reply for the CPAA980 *INQUIRY message is G.

    - Issue a Start Service Job against the child process: STRSRVJOB JOB(145778/RANDYR/CHILDAPP).

- Issue a Start Debug Command: STRDBG PGM(MYAPPLIB/CHILDAPP).
- Set whatever breakpoints are needed in CHILDAPP. When ready to continue, find the CPAA980 message and reply with G. This will unblock CHILDAPP, which allows it to run until a breakpoint is reached, at which time CHILDAPP will again stop.

  **Note:** If you reply with C to the CPAA980 message, the child process is ended before the child process' program ever receives control. In this case, on a subsequent call to **wait()** or **waitpid()**, the status information returned indicates WIFEXCEPTION(), which evaluates to a nonzero value, and WEXCEPTNUMBER() will evaluate to 0.
- The application is now stopped at a breakpoint and debugging can proceed.

13. The child's OS/400 simple job name is derived directly from the path input parameter. If path is a symbolic link to another object, the OS/400 simple job name is derived from the symbolic link itself. For example, if path was set to /QSYS.LIB/MYLIB.LIB/CHILD.PGM, the child's OS/400 simple job name would be CHILD. If /usr/bin/daughter was a symbolic link to /QSYS.LIB/MYLIB.LIB/CHILD.PGM and path was set to /usr/bin/daughter, the child's OS/400 simple job name would be DAUGHTER.

## Attributes Inherited

The child process inherits the following POSIX attributes from the parent:

1. File descriptor table (mapped according to *fd_map*).
   - If *fd_map* is NULL, all file descriptors are inherited without being reordered.

     **Note:** File descriptors that have the FD_CLOEXEC file descriptor flag set are not inherited. Refer to for additional information about the FD_CLOEXEC flag. File descriptors that are created as a result of the **opendir()** API (to implement open directory streams) are not inherited.
   - If *fd_map* is not NULL, it is a mapping from the file descriptor table of the parent process to the file descriptor table of the child process. *fd_count* specifies the number of file descriptors the child process will inherit. Except for those file descriptors designated by SPAWN_FDCLOSED, file descriptor *i* in the child process is specified by *fd_map[i]*. For example, *fd_map[5]*= 7 sets the child process' file descriptor 5 to the parent process' file descriptor 7. File descriptors *fd_count* through OPEN_MAX are closed in the child process, as are any file descriptors designated by SPAWN_FDCLOSED.

     **Note:** File descriptors that are specified in the *fd_map* array are inherited even if they have the FD_CLOEXEC file descriptor flag set. After inheritance, the FD_CLOEXEC flag in the child process' file descriptor is cleared.
   - For files descriptors that remain open, no attributes are changed.
   - If a file descriptor refers to an open instance in a file system that does not support file descriptors in two different processes pointing to the same open instance of a file, the file descriptor is closed in the child process.

     Only open files managed by the Root, QOpenSys, or user-defined file systems support inheritance of their file descriptors. All other file systems will have their file descriptors closed in the child process.
2. Process group ID

- If *inherit.flags* is set to SPAWN_SETPGROUP, the child process group ID is set to the value in *inherit.pgroup*.

  **Note:** OS/400 does not support the ability to set the process group ID for the child process to a user-specified group ID. This is a deviation from the POSIX standard.
- If *inherit.pgroup* is set to SPAWN_NEWPGROUP, the child process is put in a new process group with a process group ID equal to the process ID.
- If *inherit.pgroup* is not set to SPAWN_NEWPGROUP, the child process inherits the process group of the parent process.

If the process group that the child process is attempting to join has received the SIGKILL signal, the child process is ended.

3. Real user ID of the calling thread.
4. Real group ID of the calling thread.
5. Supplementary group IDs (group profile list) of the calling thread.
6. Current working directory of the parent process.
7. Root directory of the parent process.
8. File mode creation mask of the parent process.
9. Signal mask of the calling thread, except if the SPAWN_SETSIGMASK flag is set in *inherit.flags*. Then the child process will initially have the signal mask specified in *inherit.mask*.
10. Signal action vector, as determined by the following:
    - If the SPAWN_SETSIGDEF flag is set in *inherit.flags*, the signal specified in *inherit.sigdefault* is set to the default actions in the child process. Signals set to the default action in the parent process are set to the default action in the child process.
    - Signals set to be caught in the parent process are set to the default action in the child process.
    - Signals set to be ignored in the parent process are set to ignore in the child process, unless set to default by the above rules.
11. Priority of the parent process.

    **Note:** OS/400 prestart jobs do not inherit priority.
12. Scheduling policy (the OS/400 scheduling policy) of the parent process.
13. OS/400-specific attributes of the parent, such as job attributes, run attributes, library list, and user profile.

    **Note:** OS/400 prestart jobs inherit a subset of OS/400-specific attributes.
14. Resource limits of the parent process.

## Related Information

- The <**spawn.h**> file (see Header Files for UNIX-Type Functions)
- QlgSpawn()--Spawn Process (using NLS-enabled path name)
- spawnp()--Spawn Process with Path
- wait()--Wait for Child Process to End
- waitpid()--Wait for Specific Child Process

## Example

For an example of using this function, see Using the Spawn Process and Wait for Child Process APIs in the API Examples.

---

API introduced: V3R6

---

# spawnp()--Spawn Process with Path

Syntax

```
#include <spawn.h>

pid_t spawnp(const char                    *file,
             const int                     fd_count,
             const int                     fd_map[],
             const struct inheritance      *inherit,
             char * const                  argv[],
             char * const                  envp[]);
```

Service Program Name: QP0ZSPWN

Default Public Authority: *USE

Threadsafe: Conditional; see Usage Notes.

The **spawnp()** function creates a child process that inherits specific attributes from the parent. The attributes inherited by the child process are file descriptors, the signal mask, the signal action vector, and environment variables, among others. **spawnp()** takes the **file** parameter and searches the environment variable PATH. The **file** parameter is concatenated to each path defined in the PATH environment variable. It uses the first occurrence of the **file** parameter that is found with a mode of execute.

If the PATH environment variable does not contain a value, an error occurs. If the **file** parameter contains a "/" character, the value of **file** is used as a path and a search of the PATH or library list is not performed. Specifying a **file** parameter containing a "/" is the same as calling **spawn()**.

To search the library list, a special value for the PATH environment variable is used. The string %LIBL% can be the entire PATH value or a component of the PATH value. When the string %LIBL% is encountered, the library list is searched. For example, the following path searches the directory /usr/bin first, searches the library list next, and then searches the /tobrien/bin directory for the file:

```
PATH=/usr/bin:%LIBL%:/tobrien/bin
```

## Parameters

***file***

(Input) A file name used with the search path to find an executable file that will run in the new (child) process. The file name is expected to be in the CCSID of the job.

See QlgSpawnp()--Spawn Process with Path (using NLS-enabled file name) for a description and an example of supplying the *file* in any CCSID.

***fd_count***

(Input) The number of file descriptors the child process can inherit. It can have a value from zero to the value returned from a call to **sysconf**(_SC_OPEN_MAX).

***fd_map[]***

(Input) An array that maps the parent process file descriptor numbers to the child process file descriptor numbers. If this value is NULL, it indicates simple inheritance. **Simple inheritance** means that the child process inherits all eligible open file descriptors of the parent process. In addition, the file descriptor number in the child process is the same as the file descriptor number in the parent process. Refer to Attributes Inherited for details of file descriptor inheritance.

***inherit***

(Input) A pointer to an area of type struct inheritance. If the pointer is NULL, an error occurs. The inheritance structure contains control information to indicate attributes the child process should inherit from the parent. The following is an example of the inheritance structure, as defined in the **<spawn.h>** header file:

```
 struct inheritance {
                   flagset_t  flags;
                   int        pgroup;
                   sigset_t   sigmask;
                   sigset_t   sigdefault;
};
```

The flags field specifies the manner in which the child process should be created. Only the constants defined in **<spawn.h>** are allowed; otherwise, *spawn* returns -1 with errno set to EINVAL. The allowed constants follow:

| | |
|---|---|
| *SPAWN_SETPGROUP* | If this flag is set ON, **spawnp()** sets the process group ID of the child process to the value in pgroup. In this case, the process group field, pgroup, must be valid. If it is not valid, an error occurs. If this flag is set OFF, the pgroup field is checked to determine what the process group ID of the child process is set to. If the pgroup field is set to the constant SPAWN_NEWPGROUP, the child process group ID is set to the child process ID. If the pgroup field is not set to SPAWN_NEWPGROUP and the flags field is not set to SPAWN_SETPGROUP, the process group ID of the child process is set to the process group ID of the parent process. If the pgroup field is set to SPAWN_NEWPGROUP and the flags field is set to SPAWN_SETPGROUP, an error occurs. |
| *SPAWN_SETSIGMASK* | If this flag is set ON, **spawnp()** sets the signal blocking mask of the child process to the value in **sigmask**. In this case, the signal blocking mask must be valid. If it is not valid, an error occurs. If this flag is set OFF, **spawnp()** sets the signal blocking mask of the child process to the signal blocking mask of the calling thread. |

*SPAWN_SETSIGDEF*

If this flag is set ON, **spawnp()** sets the child process' signals identified in sigdefault to the default actions. The sigdefault must be valid. If it is not valid, an error occurs. If this flag is set OFF, **spawnp()** sets the child process' signal actions to those of the parent process. Any signals of the parent process that have a catcher specified are set to default in the child process. The child process' signal actions inherit the parent process' ignore and default signal actions.

*SPAWN_SETTHREAD_NP*

If this flag is set ON, **spawnp()** will create the child process as multithread capable. The child process will be allowed to create threads. If this flag is set OFF, the child process will not be allowed to create threads.

**Note:** The SPAWN_SETTHREAD_NP flag is a non-standard, OS/400-platform-specific extension to the inheritance structure. Applications that wish to avoid using platform-specific extensions should not use this flag.

*SPAWN_SETPJ_NP*

If this flag is set ON, **spawnp()** attempts to use available OS/400 prestart jobs. The prestart job entries that may be used follow:

- QSYS/QP0ZSPWP, if the flag SPAWN_SETTHREAD_NP is set OFF.
- QSYS/QP0ZSPWT, if the flag SPAWN_SETTHREAD_NP is set ON.

The OS/400 prestart jobs must have been started using either QSYS/QP0ZSPWP or QSYS/QP0ZSPWT as the program that identifies a prestart job entry for the OS/400 subsystem that the parent process is running under. If a prestart job entry is not defined, the child process will run as a batch immediate job under the same subsystem as the parent process.

If this flag (SPAWN_SETPJ_NP) is set OFF, the child process will run as a batch immediate job under the same subsystem as the parent process.

**Notes:**

1. In order to more closely emulate POSIX semantics, **spawnp()** will ignore the Maximum number of uses (MAXUSE) value specified for the prestart job entry. The prestart job will only be used once, behaving as if MAXUSE(1) was specified.
2. The SPAWN_SETPJ_NP flag is a non-standard, OS/400-platform-specific extension to the inheritance structure.

Applications that wish to avoid using platform-specific extensions should not use this flag.

*SPAWN_SETCOMPMSG_NP*

If this flag is set ON, **spawnp()** causes the child process to send a completion message to the user's message queue when the child process ends. If this flag is set OFF, no completion message is sent to the user's message queue when the child process ends. If both the SPAWN_SETCOMPMSG_NP and SPAWN_SETPJ_NP flags are set ON, an error occurs.

**Note:** The SPAWN_SETCOMPMSG_NP flag is a non-standard, OS/400-platform-specific extension to the inheritance structure. Applications that wish to avoid using platform-specific extensions should not use this flag.

*SPAWN_SETJOBNAMEPARENT_NP*

If this flag is set ON, **spawnp()** set the child's OS/400 simple job name to that of the parent's. If this flag is set OFF, **spawnp()** sets the child's OS/400 simple job name based on the file input parameter.

***argv[]***

(Input) An array of pointers to strings that contain the argument list for the executable file. The last element in the array must be the NULL pointer. If this parameter is NULL, an error occurs.

***envp[]***

(Input) An array of pointers to strings that contain the environment variable lists for the executable file. The last element in the array must be the NULL pointer. If this parameter is NULL, an error occurs.

## Authorities

**Figure 1-5. Authorization Required for spawnp()**

| Object Referred to | Authority Required | errno |
|---|---|---|
| Each directory in the path name preceding the executable file that will run in the new process | *X | EACCES |
| Executable file that will run in the new process | *X | EACCES |
| If executable file that will run in the new process is a shell script | *RX | EACCES |

## Return Value

| | |
|---|---|
| *value* | **spawnp()** was successful. The value returned is the process ID of the child process. |
| *-1* | **spawnp()** was not successful. The **errno** variable is set to indicate the error. |

## Error Conditions

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

*[E2BIG]*

Argument list too long.

*[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.

*[EAPAR]*

Possible APAR condition or hardware failure.

*[EBADFUNC]*

Function parameter in the signal function is not set.

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

*[EBADNAME]*

The object name specified is not correct.

*[ECANCEL]*

Operation canceled.

*[ECONVERT]*

Conversion error.

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

The specified path name is not in the CCSID of the job.

*[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.

The flags field in the inherit parameter contains an invalid value.

*[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.

*[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.

*[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.

*[ETERM]* Operation terminated.

*[ENOSYSRSC]* System resources not available to complete request.

The child process failed to start. The maximum active jobs in a subsystem may have been reached. CHGSBSD and CHGJOBQE CL commands can be used to change the maximum active jobs.

*[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

1. **spawnp()** is threadsafe, except this function will fail and errno ENOTSAFE will be set if it is called in any of the following ways:
   - From a multithreaded process and file refers to a shell script that does not exist in a threadsafe file system.
   - From a multithreaded process with a current working directory that is not in a threadsafe file system, and the PATH environment variable causes **spawnp()** to check the current working directory.
2. There are performance considerations when using **spawn()** and **spawnp()** concurrently among threads in the same process. **spawn()** and **spawnp()** serialize against other **spawn()** and **spawnp()** calls from other threads in the same process.
3. The child process is enabled for signals. A side effect of this function is that the parent process is also enabled for signals if it was not enabled for signals before this function was called.
4. If this function is called from a program running in user state <u>and</u> it specifies a system-domain program as the executable program for the child process, an exception occurs. In this case, **spawnp()** returns the process ID of the child process. On a subsequent call to **wait()** or **waitpid()**,

the status information returned indicates that an exception occurred in the child process.

5. »The program that will be run in the child process must be either a program object in the QSYS.LIB file system or an independent ASP QSYS.LIB file system (*PGM object) or a shell script (see About Shell Scripts).« The syntax of the name of the file to run must be the proper syntax for the file system in which the file resides. For example, if the program MYPROG resides in the QSYS.LIB file system and in library MYLIB, the specification for **spawnp()**. would be the following:

   ```
   MYPROG.PGM
   ```

   See QlgSpawn()--Spawn Process (using NLS-enabled path name) for an example specifying the program using the Qlg_Path_Name_T structure. The Qlg_Path_Name_T structure is supported by **QlgSpawn()** and allows the program name to be specified in any CCSID.

   **Note:** For more information about path syntaxes for the different file systems, see the Integrated File System book.

6. Spawned child processes are batch jobs or prestart jobs. As such, they do not have the ability to do 5250-type interactive I/O.

7. Spawned child processes that are OS/400 prestart jobs are similar to batch jobs. Due to the nature of prestart jobs, only the following OS/400-specific attributes are explicitly inherited in a child process when you use prestart jobs:

   - Library list
   - Language identifier
   - Country or region identifier
   - Coded character set identifier
   - Default coded character set identifier
   - Locale (as specified in the user profile)

   The child process has the same user profile as the calling thread. However, the OS/400 job attributes come from the job description specified for the prestart job entry, and the run attributes come from the class that is associated with the OS/400 subsystem used for the prestart job entry.

   **Notes:**

   1. The prestart job entry QP0ZSPWP is used with prestart jobs that will not be creating threads. The prestart job entry QP0ZSPWT is used with prestart jobs that will allow multiple threads. Both types of prestart jobs may be used in the same subsystem. The prestart job entry must be defined for the subsystem that the **spawnp()** parent process runs under in order for it to be used.

   2. The following example defines a prestart job entry (QP0ZSPWP) for use by **spawnp()** under the subsystem QINTER. The **spawnp()** API must have the SPAWN_SETPJ_NP flag set (but not SPAWN_SETTHREAD_NP) in order to use these prestart jobs:

      ```
      ADDPJE SBSD(QSYS/QINTER) PGM(QSYS/QP0ZSPWP)
             INLJOBS(20) THRESHOLD(5) ADLJOBS(5)
             JOBD(QGPL/QDFTJOBD) MAXUSE(1)
             CLS(QGPL/QINTER)
      ```

3. The following example defines a prestart job entry (QP0ZSPWT) that will create prestart jobs that are multithread capable for use by **spawnp()** under the subsystem QINTER. The **spawnp()** API must have both SPAWN_SETPJ_NP and SPAWN_SETTHREAD_NP flags set in order to use these prestart jobs. Also, the JOBD parameter must be a job description that allows multiple threads as follows:

```
ADDPJE SBSD(QSYS/QINTER) PGM(QSYS/QP0ZSPWT)
       INLJOBS(20) THRESHOLD(5) ADLJOBS(5)
       JOBD(QSYS/QAMTJOBD) MAXUSE(1)
       CLS(QGPL/QINTER)
```

Refer to the Work Management book on the V5R1 Supplemental Manuals Web site for complete details on prestart jobs.

8. Shell scripts are allowed for the child process. If a shell script is specified, the appropriate shell interpreter program is called. The shell script must be a text file and must contain the following format on the first line of the file:

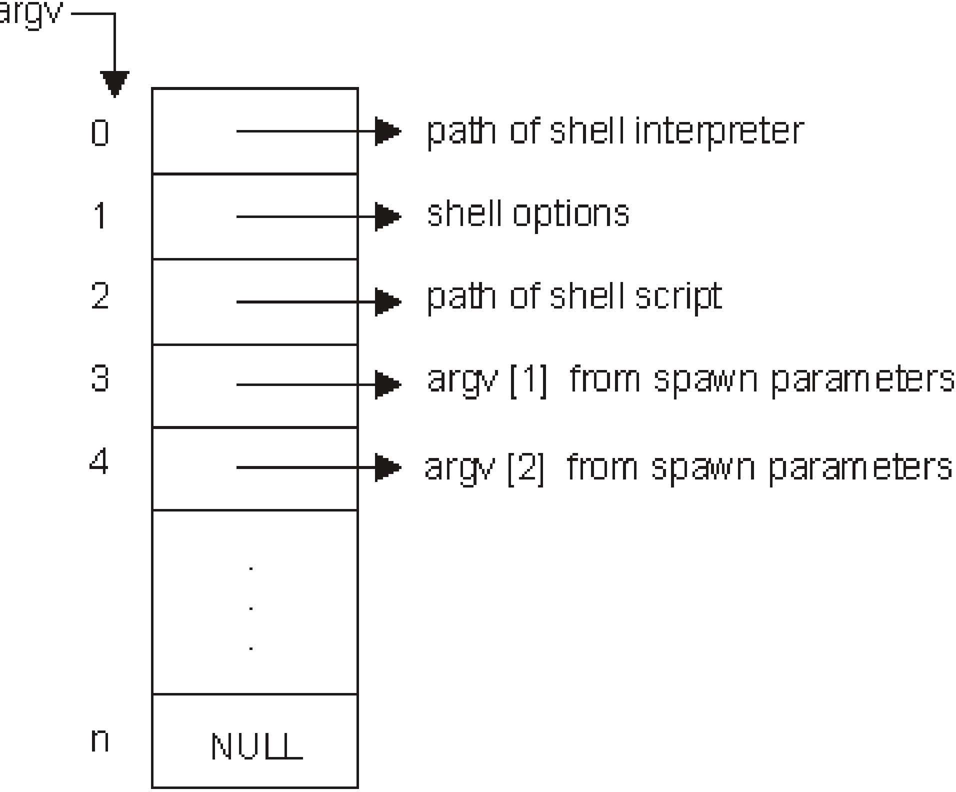
```
#!interpreter_path <options>
```

where interpreter_path is the path to the shell interpreter program.

If the calling process is multithreaded, file (the first parameter to **spawnp()**) must reference a threadsafe file system.

**spawnp()** calls the shell interpreter, passing in the shell options and the arguments passed in as a parameter to **spawnp()**. The argument list passed into the shell interpreter will look like Figure 1-6.

**Figure 1-6. Arguments to Shell Interpreter**



See About Shell Scripts for an example using **spawn()** and shell scripts.

9. Only programs that expect arguments as NULL-terminated strings can be spawned.

   The program that is run in the child's process is called at its initial entry point. The linkage to the program is C-like. The following example describes the linkage in C language terms.

   ```
   int main(int argc, char *argv[])
   [
   ]
   ```

   where the following is true:

   - **argc** is the number of arguments in **argv[]**.

   - **argv[]** is an array of arguments represented as strings. The last entry in the array is NULL. The first entry in the array, by convention, is the name of the program. **spawnp()** sets the element argv[0] to the path name of the child process' program. **spawnp()** does not move any elements of the **argv** array when it sets argv[0] to the path name of the child process' program. If that element of the array contains an argument value, the value is overwritten.

   **argv[]** is specified by the user on the interface to **spawnp()**. When **spawnp()** is called in the child's process, it passes the array to the program.

10. The child process does not inherit any of the environment variables of the parent process. That is, the default environment variable environment is empty. If the child process is to inherit all the parent process' environment variables, the extern variable **environ** can be used as the value for **envp[]** when **spawnp()** is called. If a specific set of environment variables is required in the child process, the user must build the **envp[]** array with the "name=value" strings. In the child process, **spawnp()** does the equivalent of a **putenv** on each element of the **envp[]** array. Then the extern variable **environ** will be set and available to the child process' program.

    **Note:** If the user of **spawnp()** specifies the extern variable **environ** as the **envp[]** parameter, the user must successfully call one of the following APIs before calling **spawnp()**:

    - getenv()
    - putenv()
    - Qp0zGetEnv()
    - Qp0zInitEnv()
    - Qp0zPutEnv()

    The extern variable **environ** is not initialized until one of these APIs is called in the current activation group. If **environ** is used in a call to **spawnp()** without first calling one of these APIs, **spawnp()** returns an error.

11. OS/400 handles stdin, stdout, and stderr differently than most UNIX systems. On most UNIX systems, stdin, stdout, and stderr have file descriptors 0, 1, and 2 reserved and allocated for them. On OS/400, this is not the case. There are two ramifications of this difference:

    1. File descriptor 0, 1, and 2 are allocated to the first three "files" that have descriptors allocated to them. If an application writes to file descriptor 1 assuming it is stdout, the result will not be as expected.

    2. Any API that assumes stdin, stdout, and stderr are file descriptors 0, 1, and 2 will not behave as expected.

Users and applications can enable descriptor-based standard I/O for child processes by setting environment variable QIBM_USE_DESCRIPTOR_STDIO to the value Y in the child process. This can be accomplished on the call to **spawnp()** by either of the following:

1. Specifying the extern variable **environ** as the **envp[]** parameter. This assumes that the QIBM_USE_DESCRIPTOR_STDIO environment variable exists in the calling process.

   The environment variable can be set by using one of the following:
   - API putenv("QIBM_USE_DESCRIPTOR_STDIO=Y");
   - Command ADDENVVAR ENVVAR(QIBM_USE_DESCRIPTOR_STDIO) VALUE(Y)
   - Command CHGENVVAR ENVVAR(QIBM_USE_DESCRIPTOR_STDIO) VALUE(Y)

2. Explicitly include "QIBM_USE_DESCRIPTOR_STDIO=Y" in the user-defined **envp[]** array with the "name=value" strings.

If you enable descriptor-based standard I/O for child processes, file descriptors 0, 1, and 2 are automatically used for stdin, stdout, and stderr, respectively. However, **spawnp()** must be called using a **fd_map** that has file descriptors 0, 1, and 2 properly allocated. See About Shell Scripts for an example that enables descriptor-based standard I/O for a child process. Refer to WebSphere Development Studio: ILE C/C++ Programmer's Guide for complete details on this support.

12. Spawn users have a facility to aid in debugging child processes.

    To help the user start a debug session (when **spawnp()** is the mechanism used to start the process), the user sets the environment variable QIBM_CHILD_JOB_SNDINQMSG.

    If the environment variable is assigned a numerical value, it indicates the number of descendent levels that will be enabled for debugging. This support can be used to debug applications that create children, grandchildren, great-grandchildren, and so forth. When the environment variable has a value of 1, it enables debugging of all subsequent child processes. A value of 2 enables debugging of all subsequent child processes and grandchild processes.

    When the environment variable has a value less than or equal to 0, or any non-numerical value, debugging will not occur.

    Here are the steps a user would take to debug an application by using **spawnp()**:

    Assume the user wants to debug child processes in an application called CHILDAPP found in library MYAPPLIB.

    - Set the QIBM_CHILD_JOB_SNDINQMSG environment variable to 1.

      The environment variable can be set by using one of the following:
      - API putenv("QIBM_CHILD_JOB_SNDINQMSG=1");
      - Command ADDENVVAR ENVVAR(QIBM_CHILD_JOB_SNDINQMSG) VALUE(1)
      - Command CHGENVVAR ENVVAR(QIBM_CHILD_JOB_SNDINQMSG) VALUE(1)

    - Call or run the application that specifies CHILDAPP.PGM as the fileon the **spawnp()** invocation. CHILDAPP will start running, send a CPAA980 *INQUIRY message to the user's message queue, and then will block, waiting for a reply to the message. Issue a Work

with Active Jobs (WRKACTJOB) command and find the CHILDAPP in a MSGW job status. Option 7 (Display message) performed against this job will display the CPAA980 *INQUIRY message that was sent. As part of this message, the Qualified Job Name will be displayed in the proper format to pass to the Start Service Job (STRSRVJOB) command (for example, 145778/RANDYR/CHILDAPP).

**Note:** Alternatively, a Display Messages (DSPMSG) command can be issued for the user, and the output searched for the specific CPAA980 *INQUIRY message.

**Note:** If the job's inquiry message reply specifies using the default message reply, the child process will not block since the default reply for the CPAA980 *INQUIRY message is G.

- Issue a Start Service Job against the child process: STRSRVJOB JOB(145778/RANDYR/CHILDAPP).
- Issue a Start Debug Command: STRDBG PGM(MYAPPLIB/CHILDAPP).
- Set whatever breakpoints are needed in CHILDAPP. When ready to continue, find the CPAA980 message and reply with G. This will unblock CHILDAPP, which allows it to run until a breakpoint is reached, at which time CHILDAPP will again stop.

  **Note:** If you reply with C to the CPAA980 message, the child process is ended before the child process' program ever receives control. In this case, on a subsequent call to **wait()** or **waitpid()**, the status information returned indicates WIFEXCEPTION(), which evaluates to a nonzero value, and WEXCEPTNUMBER() will evaluate to 0.
- The application is now stopped at a breakpoint and debugging can proceed.

13. The child's OS/400 simple job name is derived directly from the file input parameter. If file is a symbolic link to another object, the OS/400 simple job name is derived from the symbolic link itself. For example, if file was set to CHILD.PGM, the child's OS/400 simple job name would be CHILD. If /usr/bin/daughter was a symbolic link to /QSYS.LIB/MYLIB.LIB/CHILD.PGM, and file was set to daughter, the child's OS/400 simple job name would be DAUGHTER.

## Attributes Inherited

The child process inherits the following POSIX attributes from the parent:

1. File descriptor table (mapped according to **fd_map**).

   - If **fd_map** is NULL, all file descriptors are inherited without being reordered.

     **Note:** File descriptors that have the FD_CLOEXEC file descriptor flag set are not inherited. Refer to for additional information about the FD_CLOEXEC flag. File descriptors that are created as a result of the **opendir()** API (to implement open directory streams) are not inherited.
   - If **fd_map** is not NULL, it is a mapping from the file descriptor table of the parent process to the file descriptor table of the child process. **fd_count** specifies the number of file descriptors the child process will inherit. Except for those file descriptors designated by SPAWN_FDCLOSED, file descriptor **i** in the child process is specified by **fd_map[i]**. For example, **fd_map[5]**= 7 sets the child process' file descriptor 5 to the parent process' file descriptor 7. File descriptors **fd_count** through OPEN_MAX are closed in the child process, as are any file descriptors designated by SPAWN_FDCLOSED.

**Note:** File descriptors that are specified in the **fd_map** array are inherited even if they have the FD_CLOEXEC file descriptor flag set. After inheritance, the FD_CLOEXEC flag in the child process' file descriptor is cleared.

- For files descriptors that remain open, no attributes are changed.
- If a file descriptor refers to an open instance in a file system that does not support file descriptors in two different processes pointing to the same open instance of a file, the file descriptor is closed in the child process.

  Only open files managed by the Root, QOpenSys, or user-defined file systems support inheritance of their file descriptors. All other file systems will have their file descriptors closed in the child process.

2. Process group ID

   - If **inherit.flags** is set to SPAWN_SETPGROUP, the child process group ID is set to the value in **inherit.pgroup**.

     **Note:** OS/400 does not support the ability to set the process group ID for the child process to a user-specified group ID. This is a deviation from the POSIX standard.
   - If **inherit.pgroup** is set to SPAWN_NEWPGROUP, the child process is put in a new process group with a process group ID equal to the process ID.
   - If **inherit.pgroup** is not set to SPAWN_NEWPGROUP, the child process inherits the process group of the parent process.

   If the process group that the child process is attempting to join has received the SIGKILL signal, the child process is ended.
3. Real user ID of the calling thread.
4. Real group ID of the calling thread.
5. Supplementary group IDs (group profile list) of the calling thread.
6. Current working directory of the parent process.
7. Root directory of the parent process.
8. File mode creation mask of the parent process.
9. Signal mask of the calling thread, except if the SPAWN_SETSIGMASK flag is set in **inherit.flags**. Then the child process will initially have the signal mask specified in **inherit.mask**.
10. Signal action vector, as determined by the following:

    - If the SPAWN_SETSIGDEF flag is set in **inherit.flags**, the signal specified in **inherit.sigdefault** is set to the default actions in the child process. Signals set to the default action in the parent process are set to the default action in the child process.
    - Signals set to be caught in the parent process are set to the default action in the child

process.

  - Signals set to be ignored in the parent process are set to ignore in the child process, unless set to default by the above rules.

11. Priority of the parent process.

    **Note:** OS/400 prestart jobs do not inherit priority.
12. Scheduling policy (the OS/400 scheduling policy) of the parent process.

13. OS/400-specific attributes of the parent, such as job attributes, run attributes, library list, and user profile.

    **Note:** OS/400 prestart jobs inherit a subset of OS/400-specific attributes.
14. »Resource limits of the parent process.«

## Related Information

- The <**spawn.h**> file (see Header Files for UNIX-Type Functions)

  QlgSpawnp()--Spawn Process with Path (using NLS-enabled file name)
- spawn()--Spawn Process

- sysconf()--Get System Configuration Variables

- wait()--Wait for Child Process to End

- waitpid()--Wait for Specific Child Process

## Example

For an example of using this function, see Using the Spawn Process and Wait for Child Process APIs in Examples.

---

API introduced: V3R6

---

# »ulimit()--Get and set process limits

Syntax

```
#include <ulimit.h>

long int ulimit(int cmd, ...);
```

Service Program Name: QP0WSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **ulimit**() function provides a way to get and set process resource limits. A resource limit is a way for the operating system to enforce a limit on a variety of resources used by a process. A resource limit has a current or soft limit and a maximum or hard limit.

The **ulimit**() function is provided for compatibility with older applications. The getrlimit() and setrlimit() functions should be used for working with resource limits.

A soft limit can be changed to any value that is less than or equal to the hard limit. The hard limit can be changed to any value that is greater than or equal to the soft limit. Only a process with appropriate authorities can increase a hard limit.

The **ulimit**() function supports the following *cmd* values:

*UL_GETFSIZE (0)*
: Return the current or soft limit for the file size resource limit. The returned limit is in 512-byte blocks. The return value is the integer part of the file size resource limit divided by 512.

*UL_SETFSIZE (1)*
: Set the current or soft limit and the maximum or hard limit for the file size resource limit. The second argument is taken as a long int that represents the limit in 512-byte blocks. The specified value is multiplied by 512 to set the resource limit. If the result overflows an rlim_t, **ulimit**() returns -1 and sets *errno* to EINVAL. The new file size resource limit is returned.

## Parameters

***cmd***

(Input)

The command to be performed.

**...**

(Input)

When the *cmd* is UL_SETFSIZE, a long int that represents the limit in 512-byte blocks.

## Authorities and Locks

The current user profile must have *JOBCTL special authority to increase the hard limit.

## Return Value

**value** **ulimit()** was successful. The value is the requested limit.

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

## Error Conditions

If **ulimit()** 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] An invalid parameter was found.

An invalid *cmd* was specified.

[EPERM] Permission denied.

An attempt was made to increase the hard limit and the current user profile does not have *JOBCTL special authority.

## Related Information

- The <**ulimit.h**> file (see Header Files for UNIX-Type Functions)
- getrlimit()-Get resource limit
- setrlimit()-Set resource limit

## Example

```
#include <ulimit.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main (int argc, char *argv[])
{
  long int value;
  long int limit;

  /* Set the file size resource limit. */
  limit = 65535;
  errno = 0;
```

```
  value = ulimit(UL_SETFSIZE, limit);
  if ((value == -1) && (errno != 0)) {
    printf("ulimit() failed with errno=%d\n", errno);
    exit(1);
  }
  printf("The limit is set to %ld\n", value);

  /* Get the file size resource limit. */
  value = ulimit(UL_GETFSIZE);
  if ((value == -1) && (errno != 0)) {
    printf("ulimit() failed with errno=%d\n", errno);
    exit(1);
  }
  printf("The limit is %ld\n", value);

  exit(0);
}
```

**Example Output:**

```
The limit is set to 65535
The limit is 65535
```

«

---

Introduced: V5R2

---

# wait()--Wait for Child Process to End

Syntax

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

pid_t wait(int *stat_loc);
```

Service Program Name: QP0ZSPWN

Default Public Authority: *USE

Threadsafe: Yes

The **wait()** function suspends processing until a child process has ended. The calling thread will suspend processing until status information is available for a child process that ended. A suspended **wait()** function call can be interrupted by the delivery of a signal whose action is either to run a signal-catching function or to terminate the process. When **wait()** is successful, status information about how the child process ended (for example, whether the process ended normally) is stored in the location specified by stat_loc.

## Parameters

***stat_loc***

(Input) Pointer to an area where status information about how the child process ended is to be placed.

The status referenced by the stat_loc argument is interpreted using macros defined in the **<sys/wait.h>** header file. The macros use an argument stat_val, which is the integer value pointed to by stat_loc. When **wait()** returns with a valid process ID (pid), the macros analyze the status referenced by the stat_loc argument. The macros are as follows:

| | |
|---|---|
| *WIFEXITED(stat_val)* | Evaluates to a nonzero value if the status was returned for a child process that ended normally. |
| *WEXITSTATUS(stat_val)* | If the value of the WIFEXITED(stat_val) is nonzero, evaluates to the low-order 8 bits of the status argument that the child process passed to **exit()**, or to the value the child process returned from **main()**. |
| *WIFSIGNALED(stat_val)* | Evaluates to a nonzero value if the status was returned for a child process that ended because of the receipt of a terminating signal that was not caught by the process. |
| *WTERMSIG(stat_val)* | If the value of WIFSIGNALED(stat_val) is nonzero, evaluates to the number of the signal that caused the child process to end. |

| | |
|---|---|
| *WIFSTOPPED(stat_val)* | Evaluates to a nonzero value if the status was returned for a child process that is currently stopped. |
| *WSTOPSIG(stat_val)* | If the value of the WIFSTOPPED(stat_val) is nonzero, evaluates to the number of the signal that caused the child process to stop. |
| *WIFEXCEPTION(stat_val)* | Evaluates to a nonzero value if the status was returned for a child process that ended because of an error condition.<br><br>**Note**: The WIFEXCEPTION macro is unique to the OS/400 implementation. See the Usage Notes. |
| *WEXCEPTNUMBER(stat_val)* | If the value of the WIFEXCEPTION(stat_val) is nonzero, this macro evaluates to the last OS/400 exception number related to the child process.<br><br>**Note**: The WEXCEPTNUMBER macro is unique to the OS/400 implementation. See the Usage Notes. |

## Authorities

None

## Return Value

| | |
|---|---|
| *value* | **wait()** was successful. The value returned indicates the process ID of the child process whose status information was recorded in the storage pointed to by stat_loc. |
| *-1* | **wait()** was not successful. The errno value is set to indicate the error. |

## Error Conditions

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

| | |
|---|---|
| *[ECHILD]* | Calling process has no remaining child processes on which wait operation can be performed. |
| *[EFAULT]* | The address used for an argument is not correct.<br><br>In attempting to use an argument in a call, the system detected an address that is not valid.<br><br>While attempting to access a parameter passed to this function, the system detected an address that is not valid. |

*[EINTR]* Interrupted function call.

*[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

1. The WIFEXCEPTION macro is unique to the OS/400 implementation. This macro can be used to determine whether the child process has ended because of an exception. When WIFEXCEPTION returns a nonzero value, the value returned by the WEXCEPTNUMBER macro corresponds to the last OS/400 exception number related to the child process.

2. When a child process ends because of an exception, the ILE C run-time library catches and handles the original exception. The value returned by WEXCEPTNUMBER indicates that the exception was **CEE9901**. This is a common exception ID. If you want to determine the original exception that ended the child process, look at the job log for the child process.

3. If the child process is ended by any of the following:

    - ENDJOB OPTION(*IMMED)
    - ENDJOB OPTION(*CNTRLD) and delay time was reached
    - Debugging a child process (environment variable QIBM_CHILD_JOB_SNDINQMSG is used) and the resulting CPAA980 *INQUIRY message is replied to using C,

    then the parent's **wait()** *stat_loc* value indicates that:

    - WIFEXCEPTION(stat_val) evaluates to a nonzero value
    - WEXCEPTNUMBER(stat_val) evaluates to zero.

## Related Information

- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- The <**sys/wait.h**> file (see Header Files for UNIX-Type Functions)
- spawn()--Spawn Process
- spawnp()--Spawn Process with Path
- waitpid()--Wait for Specific Child Process
- Signal Concepts

## Example

For an example of using this function, see Using the Spawn Process and Wait for Child Process APIs in the API examples.

---

API introduced: V3R6

---

Top | UNIX-Type APIs | APIs by category

# waitpid()--Wait for Specific Child Process

Syntax

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

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Service Program Name: QP0ZSPWN

Default Public Authority: *USE

Threadsafe: Yes

The **waitpid()** function allows the calling thread to obtain status information for one of its child processes. The calling thread suspends processing until status information is available for the specified child process, if the options argument is 0. A suspended **waitpid()** function call can be interrupted by the delivery of a signal whose action is either to run a signal-catching function or to terminate the process. When **waitpid()** is successful, status information about how the child process ended (for example, whether the process ended normally) is stored in the location specified by stat_loc.

The **waitpid()** function behaves the same as **wait()** if the pid argument is (pid_t)-1 and the options argument is 0.

## Parameters

***pid***

(Input) A process ID or a process group ID to identify the child process or processes on which **waitpid()** should operate.

***stat_loc***

(Input) Pointer to an area where status information about how the child process ended is to be placed.

***options***

(Input) An integer field containing flags that define how **waitpid()** should operate.

The pid argument specifies a set of child processes for which status is requested. The **waitpid()** function only returns the status of a child process from the following set:

- If pid is equal to (pid_t)-1, status is requested for any child process. In this respect, **waitpid()** is then equivalent to **wait()**.
- If pid is greater than (pid_t)0, it specifies the process ID of a single child process for which status is requested.
- If pid is (pid_t)0, status is requested for any child process whose process group ID is equal to that

of the calling thread.

- If pid is less than (pid_t)-1, status is requested for any child process whose process group ID is equal to the absolute value of pid.

The status referenced by the stat_loc argument is interpreted using macros defined in the **<sys/wait.h>** header file. The macros use an argument stat_val, which is the integer value pointed to by stat_loc. When **waitpid()** returns with a valid process ID (pid), the macros analyze the status referenced by the stat_loc argument. The macros are as follows:

*WIFEXITED(stat_val)*
: Evaluates to a nonzero value if the status was returned for a child process that ended normally.

*WEXITSTATUS(stat_val)*
: If the value of the WIFEXITED(stat_val) is nonzero, evaluates to the low-order 8 bits of the status argument that the child process passed to **exit()**, or to the value the child process returned from **main()**.

*WIFSIGNALED(stat_val)*
: Evaluates to a nonzero value if the status was returned for a child process that ended because of the receipt of a terminating signal that was not caught by the process.

*WTERMSIG(stat_val)*
: If the value of WIFSIGNALED(stat_val) is nonzero, evaluates to the number of the signal that caused the child process to end.

*WIFSTOPPED(stat_val)*
: Evaluates to a nonzero value if the status was returned for a child process that is currently stopped.

*WSTOPSIG(stat_val)*
: If the value of the WIFSTOPPED(stat_val) is nonzero, evaluates to the number of the signal that caused the child process to stop.

*WIFEXCEPTION(stat_val)*
: Evaluates to a nonzero value if the status was returned for a child process that ended because of an error condition.

  **Note**: The WIFEXCEPTION macro is unique to the OS/400 implementation. See the Usage Notes.

*WEXCEPTNUMBER(stat_val)*
: If the value of the WIFEXCEPTION(stat_val) is nonzero, this macro evaluates to the last OS/400 exception number related to the child process.

  **Note**: The WEXCEPTNUMBER macro is unique to the OS/400 implementation. See the Usage Notes.

The options argument can be set to either 0 or WNOHANG. WNOHANG indicates that the **waitpid()** function should not suspend processing of the calling thread if status is not immediately available for one of the child processes specified by pid. If WNOHANG is specified and no child process is immediately available, **waitpid()** returns 0.

## Authorities

None

## Return Value

| | |
|---|---|
| *value* | **waitpid()** was successful. The value returned indicates the process ID of the child process whose status information was recorded in the storage pointed to by stat_loc. |
| *0* | WNOHANG was specified on the options parameter, but no child process was immediately available. |
| *-1* | **waitpid()** was not successful. The errno value is set to indicate the error. |

## Error Conditions

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

*[ECHILD]*
: Calling process has no remaining child processes on which wait operation can be performed.

*[EINVAL]*
: An invalid parameter was found.

  A parameter passed to this function is not valid.

*[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.

*[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

1. The WIFEXCEPTION macro is unique to the OS/400 implementation. This macro can be used to determine whether the child process has ended because of an exception. When WIFEXCEPTION returns a nonzero value, the value returned by the WEXCEPTNUMBER macro corresponds to the last OS/400 exception number related to the child process.

2. When a child process ends because of an exception, the ILE C run-time library catches and handles the original exception. The value returned by WEXCEPTNUMBER indicates that the exception was **CEE9901**. This is a common exception ID. If you want to determine the original exception that ended the child process, look at the job log for the child process.

3. If the child process is ended by any of the following:
   - ENDJOB OPTION(*IMMED),
   - ENDJOB OPTION(*CNTRLD) and delay time was reached, or
   - Debugging a child process (environment variable QIBM_CHILD_JOB_SNDINQMSG is used) and the resulting CPAA980 *INQUIRY message is replied to using C,

   then the parent's **wait()** *stat_loc* value indicates that:
   - WIFEXCEPTION(stat_val) evaluates to a nonzero value, and
   - WEXCEPTNUMBER(stat_val) evaluates to zero.

## Related Information

- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- The <**sys/wait.h**> file (see Header Files for UNIX-Type Functions)
- spawn()--Spawn Process
- spawnp()--Spawn Process with Path
- wait()--Wait for Child Process to End
- Signal Concepts

## Example

For an example of using this function, see Using the Spawn Process and Wait for Child Process APIs in API examples.

---

API introduced: V3R6

---

# About Shell Scripts

A **shell** (or shell interpreter) is a command interpreter. The shell interprets text strings and performs some function for each string. As part of interpreting the string, the shell may do variable or wildcard replacement or change the string in some way. Typically, the shell itself performs functions specified by internal commands and spawns a child process to perform processing on the external commands. Depending on the command, the shell then does one of the following:

- Waits for the child process to complete
- Continues processing with the next command

A **shell script** is a text file whose format defines the following:

- A shell interpreter (path and program)
- Options or arguments to pass to the shell
- Text to be interpreted as a series of commands to the shell

The format of a shell script, starting on line one and column one, is as follows:

```
#!interpreter_path <options>
text to be interpreted
text to be interpreted
 .
 .
 .
```

where

interpreter_path is the shell interpreter.

options are the options to pass to the shell interpreter.

The **spawn()** and **spawnp()** functions support shell scripts. OS/400 currently provides the Qshell Interpreter. The Qshell Interpreter is a standard command interpreter for OS/400 based on the POSIX 1003.2 standard and X/Open CAE Specification for Shell and Utilities.

**Examples**

The following is an example of using **spawn()** to run a shell script written for the Qshell Interpreter:

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

int main(int argc, char *argv[])
{
   int fd_map[3], stdoutFds[2];
   char *xmp_argv[4], *xmp_envp[3];
   struct inheritance xmp_inherit = {0};
   char buffer[20];
   pid_t child_pid, wait_rv;
   int wait_stat_loc, rc;

   xmp_argv[0] = "/home/myuserid/myscript";
   xmp_argv[1] = "Hello";
   xmp_argv[2] = "world!";
   xmp_argv[3] = NULL;

   xmp_envp[0] =
       "NLSPATH=/QIBM/ProdData/OS400/Shell/MRI2924/%N";
   xmp_envp[1] = "QIBM_USE_DESCRIPTOR_STDIO=Y";
   xmp_envp[2] = NULL;

   if (pipe(stdoutFds) != 0) {
       printf("failure on pipe\n");
       return 1;
   }
```

```
    fd_map[0] = stdoutFds[1];
    fd_map[1] = stdoutFds[1];
    fd_map[2] = stdoutFds[1];

    if ((child_pid = spawn("/home/myuserid/myscript", 3,
                           fd_map, &xmp_inherit, xmp_argv,
                           xmp_envp)) == -1) {
        printf("failure on spawn\n");
        return 1;
    }

    if ((wait_rv = waitpid(child_pid,
                            &wait_stat_loc, 0)) == -1) {
        printf("failure on waitpid\n");
        return 1;
    }
    close(stdoutFds[1]);

    while ((rc = read(stdoutFds[0],
                 buffer, sizeof(buffer))) > 0) {
        buffer[rc] = '\0';
        printf("%s", buffer);
    }
    close(stdoutFds[0]);
    return 0;
}
```

where "/home/myuserid/myscript" could look like the following:

```
    #!/usr/bin/qsh
    print $1 $2
```

**Example Output:**

```
    Hello world!
```

---

# 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. |

---