



iSeries

UNIX-Type -- Interprocess Communication (IPC) APIs

*Version 5 Release 3*







@server

iSeries

UNIX-Type -- Interprocess Communication (IPC) APIs

*Version 5 Release 3*

**Note**

Before using this information and the product it supports, be sure to read the information in "Notices," on page 153.

**Sixth Edition (August 2005)**

This edition applies to version 5, release 3, modification 0 of Operating System/400 (product number 5722-SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

© Copyright International Business Machines Corporation 1998, 2005. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents

<b>Interprocess Communication (IPC) APIs</b>	<b>1</b>
APIs	3
ftok()—Generate IPC Key from File Name	3
Parameters	3
Authorities	4
Return Value	4
Error Conditions	4
Usage Notes	6
Related Information	7
Example	7
msgctl()—Perform Message Control Operations	8
Parameters	9
Authorities	9
Return Value	10
Error Conditions	10
Error Messages	11
Usage Notes	11
Related Information	11
Example	11
msgget()—Get Message Queue	11
Parameters	12
Authorities	12
Return Value	13
Error Conditions	13
Error Messages	14
Usage Notes	14
Related Information	14
Example	14
msgrcv()—Receive Message Operation	15
Parameters	16
Authorities	16
Return Value	16
Error Conditions	16
Error Messages	18
Usage Notes	18
Related Information	18
Example	18
msgsnd()—Send Message Operation	19
Parameters	20
Authorities	20
Return Value	20
Error Conditions	20
Error Messages	21
Usage Notes	21
Related Information	22
Example	22
QlgFtok()—Generate IPC Key from File Name (using NLS-enabled path name)	23
Parameters	23
Related Information	23
Example	23
QlgSem_open()—Open Named Semaphore (using NLS-enabled path name)	24
Parameters	25
Error Conditions	25
Related Information	25
Example	25
QlgSem_open_np()—Open Named Semaphore with Maximum Value (using NLS-enabled path name)	26
Parameters	27
Error Conditions	27
Related Information	27
Example	27
QlgSem_unlink()—Unlink Named Semaphore (using NLS-enabled path name)	28
Parameters	28
Error Conditions	28
Related Information	29
Example	29
Delete Interprocess Communication Objects	
(QP0ZDIPC) API	30
Authorities and Locks	30
Required Parameter Group	30
Delete Control Format	30
Field Descriptions	31
Error Messages	31
Open List of Interprocess Communication Objects	
(QP0ZOLIP) API	32
Authorities and Locks	33
Required Parameter Group	33
FIPC0100 Format	34
Field Descriptions	34
LSST0100 Format	35
LMSQ0100 Format	36
LSHM0100 Format	36
LNSM0100 Format	37
LUSM0100 Format	38
Field Descriptions	39
Error Messages	43
Open List of Semaphores (QP0ZOLSM) API	44
Authorities and Locks	44
Required Parameter Group	45
LSEM0100 Format	45
Field Descriptions	46
Error Messages	47
Retrieve an Interprocess Communication Object	
(QP0ZRIPC) API	47
Authorities and Locks	48
Required Parameter Group	48
RSST0100 Format	48
RMSQ0100 Format	49
RSHM0100 Format	50
Field Descriptions	51
Error Messages	56
semctl()—Perform Semaphore Control Operations	57
Parameters	58
Authorities	59
Return Value	60
Error Conditions	60
Error Messages	61
Usage Notes	61
Related Information	61

Example . . . . .	61	Parameters . . . . .	78
semget()—Get Semaphore Set with Key . . . . .	62	Authorities . . . . .	79
Authorities . . . . .	63	Return Value . . . . .	79
Return Value . . . . .	63	Error Conditions. . . . .	79
Error Conditions. . . . .	63	Error Messages . . . . .	80
Error Messages . . . . .	64	Related Information . . . . .	80
Usage Notes . . . . .	64	Example . . . . .	80
Related Information . . . . .	65	sem_open_np()—Open Named Semaphore with	
Example . . . . .	65	Maximum Value. . . . .	81
semop()—Perform Semaphore Operations on		Parameters . . . . .	81
Semaphore Set . . . . .	65	Authorities . . . . .	82
Parameters . . . . .	66	Return Value . . . . .	83
Authorities . . . . .	66	Error Conditions. . . . .	83
Return Value . . . . .	67	Error Messages . . . . .	84
Error Conditions. . . . .	67	Related Information . . . . .	84
Error Messages . . . . .	68	Example . . . . .	84
Usage Notes . . . . .	68	sem_post()—Post to Semaphore . . . . .	85
Related Information . . . . .	68	Parameters . . . . .	85
Example . . . . .	68	Authorities . . . . .	85
sem_close()—Close Named Semaphore . . . . .	69	Return Value . . . . .	85
Parameters . . . . .	69	Error Conditions. . . . .	85
Authorities . . . . .	69	Error Messages . . . . .	85
Return Value . . . . .	69	Related Information . . . . .	85
Error Conditions. . . . .	69	Example . . . . .	86
Error Messages . . . . .	69	Output:. . . . .	86
Related Information . . . . .	70	sem_post_np()—Post Value to Semaphore . . . . .	86
Example . . . . .	70	Parameters . . . . .	87
sem_destroy()—Destroy Unnamed Semaphore. . . . .	70	Authorities . . . . .	87
Parameters . . . . .	71	Return Value . . . . .	87
Authorities . . . . .	71	Error Conditions. . . . .	87
Return Value . . . . .	71	Error Messages . . . . .	87
Error Conditions. . . . .	71	Related Information . . . . .	87
Error Messages . . . . .	71	Example . . . . .	88
Related Information . . . . .	71	Output:. . . . .	88
Example . . . . .	71	sem_trywait()—Try to Decrement Semaphore . . . . .	89
sem_getvalue()—Get Semaphore Value . . . . .	72	Parameters . . . . .	89
Authorities . . . . .	72	Authorities . . . . .	89
Return Value . . . . .	72	Return Value . . . . .	89
Error Conditions. . . . .	72	Error Conditions. . . . .	89
Error Messages . . . . .	73	Error Messages . . . . .	89
Related Information . . . . .	73	Related Information . . . . .	90
Example . . . . .	73	Example . . . . .	90
Output:. . . . .	73	Output:. . . . .	90
sem_init()—Initialize Unnamed Semaphore. . . . .	74	sem_unlink()—Unlink Named Semaphore . . . . .	91
Parameters . . . . .	74	Parameters . . . . .	91
Authorities . . . . .	74	Authorities . . . . .	91
Return Value . . . . .	74	Return Value . . . . .	91
Error Conditions. . . . .	74	Error Conditions. . . . .	92
Error Messages . . . . .	75	Error Messages . . . . .	92
Related Information . . . . .	75	Related Information . . . . .	92
Example . . . . .	75	Example . . . . .	92
sem_init_np()—Initialize Unnamed Semaphore with		sem_wait()—Wait for Semaphore . . . . .	93
Maximum Value. . . . .	75	Parameters . . . . .	93
Parameters . . . . .	76	Authorities . . . . .	93
Authorities . . . . .	76	Return Value . . . . .	93
Return Value . . . . .	76	Error Conditions. . . . .	93
Error Conditions. . . . .	76	Error Messages . . . . .	93
Error Messages . . . . .	77	Related Information . . . . .	93
Related Information . . . . .	77	Example . . . . .	94
Example . . . . .	77	Output:. . . . .	94
sem_open()—Open Named Semaphore . . . . .	78	sem_wait_np()—Wait for Semaphore with Timeout . . . . .	95

Parameters . . . . .	95	Exit Programs . . . . .	111
Authorities . . . . .	95	Integrated File System Scan on Close Exit Program	111
Return Value . . . . .	95	Restrictions . . . . .	112
Error Conditions. . . . .	95	Authorities and Locks . . . . .	113
Error Messages . . . . .	96	Program Data . . . . .	113
Related Information . . . . .	96	Required Parameter Group . . . . .	113
Example . . . . .	96	Format of Integrated File System Close Exit	
Output. . . . .	97	Information (Input) . . . . .	113
shmatt()—Attach Shared Memory Segment to		Format of Status Information (Output) . . . . .	114
Current Process . . . . .	97	Field Descriptions . . . . .	114
Parameters . . . . .	98	Usage Notes. . . . .	119
Authorities . . . . .	98	Related Information . . . . .	120
Return Value . . . . .	98	Integrated File System Scan on Open Exit Program	121
Error Conditions. . . . .	98	Restrictions . . . . .	122
Error Messages . . . . .	100	Authorities and Locks . . . . .	122
Usage Notes. . . . .	100	Program Data . . . . .	122
Related Information . . . . .	100	Required Parameter Group . . . . .	122
Example . . . . .	100	Format of Integrated File System Open Exit	
shmctl()—Perform Shared Memory Control		Information (Input) . . . . .	123
Operations . . . . .	101	Format of Status Information (Output) . . . . .	123
Parameters . . . . .	101	Field Descriptions . . . . .	124
Authorities . . . . .	102	Scan Key List and Scan Key Signatures. . . . .	128
Return Value . . . . .	102	Coded Character Set Identifier (CCSID)	
Error Conditions . . . . .	103	Information . . . . .	129
Error Messages . . . . .	104	Usage Notes. . . . .	130
Usage Notes. . . . .	104	Related Information . . . . .	131
Related Information . . . . .	104	Process a Path Name Exit Program . . . . .	131
Example . . . . .	104	Parameters . . . . .	132
shmddt()—Detach Shared Memory Segment from		Save Storage Free Exit Program . . . . .	133
Calling Process . . . . .	104	Required Parameter Group . . . . .	134
Parameters . . . . .	105	Related Information . . . . .	134
Authorities . . . . .	105	Concepts . . . . .	135
Return Value . . . . .	105	Identifier Based Services. . . . .	135
Error Conditions . . . . .	105	Message Queues . . . . .	136
Error Messages . . . . .	106	Semaphore Sets. . . . .	138
Usage Notes. . . . .	106	Shared Memory . . . . .	140
Related Information . . . . .	106	Pointer Based Services . . . . .	142
Example . . . . .	106	Managing IPC Objects . . . . .	144
shmget()—Get ID of Shared Memory Segment with		Header Files for UNIX-Type Functions . . . . .	144
Key . . . . .	106	Errno Values for UNIX-Type Functions . . . . .	147
Parameters . . . . .	107	<b>Appendix. Notices . . . . .</b>	<b>153</b>
Authorities . . . . .	108	Trademarks . . . . .	154
Return Value . . . . .	109	Terms and conditions for downloading and	
Error Conditions . . . . .	109	printing publications . . . . .	155
Error Messages . . . . .	110	Code disclaimer information . . . . .	156
Usage Notes. . . . .	110		
Related Information . . . . .	110		
Example . . . . .	111		





---

## Interprocess Communication (IPC) APIs

Interprocess communication (IPC) on OS/400<sup>(R)</sup> is made up of five services divided into the two categories of identifier-based services and pointer-based services. The identifier-based IPC services consist of message queues, semaphore sets, and shared memory. The pointer-based services consist of unnamed and named semaphores. The basic purpose of these services is to provide OS/400 processes and threads with a way to communicate with each other through a set of standard APIs. These functions are based on the definitions in the Single UNIX<sup>(R)</sup> Specification.

For additional information on the Interprocess Communication APIs, see:

- “Identifier Based Services” on page 135
  - “Message Queues” on page 136
  - “Semaphore Sets” on page 138
  - “Shared Memory” on page 140
- “Pointer Based Services” on page 142 (Named and Unnamed Semaphores)
- “Managing IPC Objects” on page 144

The interprocess communication functions and what they do are:

- “ftok()—Generate IPC Key from File Name” on page 3 (Generate IPC Key from File Name) generates an IPC key based on the combination of path and id.
- “msgctl()—Perform Message Control Operations” on page 8 (Perform Message Control Operations) provides message control operations as specified by cmd on the message queue specified by msqid.
- “msgget()—Get Message Queue” on page 11 (Get Message Queue) returns the message queue identifier associated with the parameter key.
- “msgrcv()—Receive Message Operation” on page 15 (Receive Message Operation) reads a message from the queue associated with the message queue identifier specified by msqid and places it in the user-defined buffer pointed to by msgp.
- “msgsnd()—Send Message Operation” on page 19 (Send Message Operation) is used to send a message to the queue associated with the message queue identifier specified by msqid.
- “QlgFtok()—Generate IPC Key from File Name (using NLS-enabled path name)” on page 23 (Generate IPC Key from File Name (using NLS-enabled path name)) generates an IPC key based on the combination of path and id.
- “QlgSem\_open()—Open Named Semaphore (using NLS-enabled path name)” on page 24 (Open Named Semaphore (using NLS-enabled path name)) opens a named semaphore and returns a semaphore pointer that may be used on subsequent calls to sem\_post(), sem\_post\_np(), sem\_wait(), sem\_wait\_np(), sem\_trywait(), sem\_getvalue(), and sem\_close().
- “QlgSem\_open\_np()—Open Named Semaphore with Maximum Value (using NLS-enabled path name)” on page 26 (Open Named Semaphore with Maximum Value (using NLS-enabled path name)) opens a named semaphore and returns a semaphore pointer that may be used on subsequent calls to sem\_post(), sem\_post\_np(), sem\_wait(), sem\_wait\_np(), sem\_trywait(), sem\_getvalue(), and sem\_close().
- “QlgSem\_unlink()—Unlink Named Semaphore (using NLS-enabled path name)” on page 28 (Unlink Named Semaphore (using NLS-enabled path name)) unlinks a named semaphore.
- “Delete Interprocess Communication Objects (QP0ZDIPC) API” on page 30 (Delete Interprocess Communication Objects) deletes one or more interprocess communication (IPC) objects as specified by the delete control parameter.
- “Open List of Interprocess Communication Objects (QP0ZOLIP) API” on page 32 (Open List of Interprocess Communication Objects) lets you generate a list of interprocess communication (IPC) objects and descriptive information based on the selection parameters.

- “Open List of Semaphores (QP0ZOLSM) API” on page 44 (Open List of Semaphores) lets you generate a list of description information about the semaphores within a semaphore set.
- “Retrieve an Interprocess Communication Object (QP0ZRIPC) API” on page 47 (Retrieve an Interprocess Communication Object) lets you generate detailed information about a single interprocess communication (IPC) object.
- “semctl()—Perform Semaphore Control Operations” on page 57 (Perform Semaphore Control Operations) provides semaphore control operations as specified by `cmd` on the semaphore specified by `semnum` in the semaphore set specified by `semid`.
- “semget()—Get Semaphore Set with Key” on page 62 (Get Semaphore Set with Key) returns the semaphore ID associated with the specified semaphore key.
- “semop()—Perform Semaphore Operations on Semaphore Set” on page 65 (Perform Semaphore Operations on Semaphore Set) performs operations on semaphores in a semaphore set. These operations are supplied in a user-defined array of operations.
- “sem\_close()—Close Named Semaphore” on page 69 (Close Named Semaphore) closes a named semaphore that was previously opened by a thread of the current process using `sem_open()` or `sem_open_np()`.
- “sem\_destroy()—Destroy Unnamed Semaphore” on page 70 (Destroy Unnamed Semaphore) destroys an unnamed semaphore that was previously initialized using `sem_init()` or `sem_init_np()`.
- “sem\_getvalue()—Get Semaphore Value” on page 72 (Get Semaphore Value) retrieves the value of a named or unnamed semaphore.
- “sem\_init()—Initialize Unnamed Semaphore” on page 74 (Initialize Unnamed Semaphore) initializes an unnamed semaphore and sets its initial value.
- “sem\_init\_np()—Initialize Unnamed Semaphore with Maximum Value” on page 75 (Initialize Unnamed Semaphore with Maximum Value) initializes an unnamed semaphore and sets its initial value.
- “sem\_open()—Open Named Semaphore” on page 78 (Open Named Semaphore) opens a named semaphore, returning a semaphore pointer that may be used on subsequent calls to `sem_post()`, `sem_post_np()`, `sem_wait()`, `sem_wait_np()`, `sem_trywait()`, `sem_getvalue()`, and `sem_close()`.
- “sem\_open\_np()—Open Named Semaphore with Maximum Value” on page 81 (Open Named Semaphore with Maximum Value) opens a named semaphore, returning a semaphore pointer that may be used on subsequent calls to `sem_post()`, `sem_post_np()`, `sem_wait()`, `sem_wait_np()`, `sem_trywait()`, `sem_getvalue()`, and `sem_close()`.
- “sem\_post()—Post to Semaphore” on page 85 (Post to Semaphore) posts to a semaphore, incrementing its value by one.
- “sem\_post\_np()—Post Value to Semaphore” on page 86 (Post Value to Semaphore) posts to a semaphore, incrementing its value by the increment specified in the options parameter.
- “sem\_trywait()—Try to Decrement Semaphore” on page 89 (Try to Decrement Semaphore) attempts to decrement the value of the semaphore.
- “sem\_unlink()—Unlink Named Semaphore” on page 91 (Unlink Named Semaphore) unlinks a named semaphore.
- “sem\_wait()—Wait for Semaphore” on page 93 (Wait for Semaphore) decrements by one the value of the semaphore.
- “sem\_wait\_np()—Wait for Semaphore with Timeout” on page 95 (Wait for Semaphore with Timeout) attempts to decrement by one the value of the semaphore.
- “shmat()—Attach Shared Memory Segment to Current Process” on page 97 (Attach Shared Memory Segment to Current Process) returns the address of the shared memory segment associated with the specified shared memory identifier.
- “shmctl()—Perform Shared Memory Control Operations” on page 101 (Perform Shared Memory Control Operations) provides shared memory control operations as specified by `cmd` on the shared memory segment specified by `shmid`.

- “`shmdt()`—Detach Shared Memory Segment from Calling Process” on page 104 (Detach Shared Memory Segment from Calling Process) detaches the shared memory segment specified by `shmaddr` from the calling process.
- “`shmget()`—Get ID of Shared Memory Segment with Key” on page 106 (Get ID of Shared Memory Segment with Key) returns the shared memory ID associated with the specified shared memory key.

**Note:** These functions use header (include) files from the library `QSYSINC`, which is optionally installable. Make sure `QSYSINC` is installed on your system before using any of the functions. See “Header Files for UNIX-Type Functions” on page 144 for the file and member name of each header file.

Top | UNIX-Type APIs | APIs by category

---

## APIs

These are the APIs for this category.

---

### `ftok()`—Generate IPC Key from File Name

Syntax

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

```
key_t ftok(const char *path, int id);
```

Service Program Name: QP0ZCPA

Default Public Authority: \*USE

Threadsafe: Conditional; see “Usage Notes” on page 6.

The `ftok()` function generates an IPC key based on the combination of *path* and *id*.

Identifier-based interprocess communication facilities require you to supply a key to the `msgget()`, `semget()`, `shmget()` subroutines to obtain interprocess communication identifiers. The `ftok()` function is one mechanism to generate these keys.

If the values for *path* and *id* are the same as a previous call to `ftok()` and the file named by *path* was not deleted and re-created in between calls to `ftok()`, `ftok()` will return the same key.

The `ftok()` function returns different keys if different values of *path* and *id* are used.

Only the low-order 8-bits of *id* are significant. The remaining bits are ignored by `ftok()`.

## Parameters

**path** (Input) The path name of the file used in combination with *id* to generate the key.

See “`QlgFtok()`—Generate IPC Key from File Name (using NLS-enabled path name)” on page 23 for a description and an example of supplying the *path* in any CCSID.

**id** (Input) The integer identifier used in combination with *path* to generate the key. Only the low order 8-bits of *id* are significant. The remaining bits will be ignored.

## Authorities

### Authorization Required for `ftok()` (excluding `QOPT`)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object	None	None

### Authorization Required for `ftok()` in the `QOPT` File System

Object Referred to	Authority Required	errno
Volume containing directory or object	*USE	EACCES
Directory or object within volume	None	None

## Return Value

*value*                    `ftok()` was successful.  
(*key\_t*)-1                `ftok()` was not successful. The *errno* variable is set to indicate the error.

## Error Conditions

If `ftok()` is not successful, *errno* indicates one of the following errors.

### [EACCES]

Permission denied.

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

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

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

### [EAGAIN]

Operation would have caused the process to be suspended.

### [EBADFID]

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

The file ID table is missing or damaged.

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

### [EBADNAME]

The object name specified is not correct.

### [EBUSY]

Resource busy.

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

*[ECONVERT]*

Conversion error.

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

*[EDAMAGE]*

A damaged object was encountered.

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

*[EFAULT]*

The address used for an argument is not correct.

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

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

*[EFILECVT]*

File ID conversion of a directory failed.

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

*[EINTR]*

Interrupted function call.

*[EINVAL]*

The value specified for the argument is not correct.

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

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

*[EIO]*

Input/output error.

A physical I/O error occurred.

A referenced object may be damaged.

*[ELOOP]*

A loop exists in the symbolic links.

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

*[ENAMETOOLONG]*

A path name is too long.

A path name is longer than PATH\_MAX characters or some component of the name is longer than NAME\_MAX characters while \_POSIX\_NO\_TRUNC is in effect. For symbolic links, the length of the name string substituted for a symbolic link exceeds PATH\_MAX. The PATH\_MAX and NAME\_MAX values can be determined using the **pathconf()** function.

*[ENOENT]*

No such path or directory.

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

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

*[ENOMEM]*

Storage allocation request failed.

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

There is not enough memory to perform the requested function.

*[ENOSPC]*

No space available.

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

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

*[ENOTDIR]*

Not a directory.

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

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

*[ENOTSAFE]*

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

*[ENOTSUP]*

Operation not supported.

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

*[EPERM]*

Operation not permitted.

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

*[EROOBB]*

Object is read only.

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

*[ESTALE]*

File or object handle rejected by server.

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

*[EUNKNOWN]*

Unknown system state.

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

## Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:

- Where multiple threads exist in the job.
  - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
    - Root
    - QOpenSys
    - User-defined
    - QNTC
    - QSYS.LIB
    - Independent ASP QSYS.LIB
    - QOPT
    - [» Network File System «](#)
    - [» QFileSvr.400 «](#)
2. If the values for *path* and *id* are the same as a previous call to **ftok()** and if the file named by *path* was deleted and re-created in between calls to **ftok()**, **ftok()** will return a different key.
  3. The **ftok()** function will return the same key for different values of *path* if the path names refer to symbolic links or hard links whose target files are the same.
  4. The **ftok()** function may return the same key for different values of *path* if the target files are in different file systems.
  5. The **ftok()** function may return the same key for different values of *path* if the target file is in a file system that contains more than  $2^{24}$  files.

## Related Information

- “QlgFtok()—Generate IPC Key from File Name (using NLS-enabled path name)” on page 23—Generate IPC Key from File Name (using NLS-enabled path name)
- “msgget()—Get Message Queue” on page 11—Get Message Queue
- “semget()—Get Semaphore Set with Key” on page 62—Get Semaphore Set with Key
- “shmget()—Get ID of Shared Memory Segment with Key” on page 106—Get ID of Shared Memory Segment with Key

## Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **ftok()** and **semget()** functions.

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    key_t myKey;
    int  semid;

    /* Use ftok to generate a key associated with a file. */
    /* Every process will get the same key back if the */
    /* caller calls with the same parameters.          */
    myKey = ftok("/myApplication/myFile", 42);
    if(myKey == -1) {
        printf("ftok failed with errno = %d\n", errno);
        return -1;
    }
}
```

```

/* Call an xxxget() API, where xxx is sem, shm, or msg. */
/* This will create or reference an existing IPC object */
/* with the 'well known' key associated with the file */
/* name used above. */
semid = semget(myKey, 1, 0666 | IPC_CREAT);
if(semid == -1) {
    printf("semget failed with errno = %d\n", errno);
    return -1;
}

/* ... Use the semaphore as required ... */
return 0;
}

```

API introduced: V4R3

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

---

## msgctl()—Perform Message Control Operations

### Syntax

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

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Service Program Name: QP0ZUSHR

Default Public Authority: \*USE

Threadsafe: Yes

The **msgctl()** function allows the caller to control the message queue specified by the *msqid* parameter.

A message queue is controlled by setting the *cmd* parameter to one of the following values:

### IPC\_RMID (0x00000000)

Remove the message queue identifier *msqid* from the system and destroy any messages on the message queue. Any threads that are waiting in **msgsnd()** or **msgrcv()** are woken up and **msgsnd()** or **msgrcv()** returns with a return value of -1 and *errno* set to EIDRM.

The IPC\_RMID command can be run only by a thread with appropriate privileges or one that has an effective user ID equal to the user ID of the owner or the user ID of the creator of the message queue. The structure pointed to by *\*buf* is ignored and can be NULL.

### IPC\_SET (0x00000001)

Set the user ID of the owner, the group ID of the owner, the permissions, and the maximum number of bytes for the message queue to the values in the *msg\_perm.uid*, *msg\_perm.gid*, *msg\_perm.mode*, and *msg\_qbytes* members of the *msqid\_ds* data structure pointed to by *\*buf*.

The IPC\_SET command can be run only by a thread with appropriate privileges or one that has an effective user ID equal to the user ID of the owner or the user ID of the creator of the message queue.

In addition, only a thread with appropriate privileges can increase the maximum number of bytes for the message queue.



## IPC\_STAT (0x00000002)

Store the current value of each member of the `msqid_ds` data structure into the structure pointed to by `*buf`. The `IPC_STAT` command requires read permission to the message queue.

## Parameters

- msqid** (Input) Message queue identifier, a positive integer. It is returned by the “`msgget()`—Get Message Queue” on page 11 function and used to identify the message queue on which to perform the control operation.
- cmd** (Input) Command, the control operation to perform on the message queue. Valid values are listed above.
- buf** (I/O) Pointer to the message queue data structure to be used to get or set message queue information.

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

<i>struct ipc_perm</i>	The members of the <code>ipc_perm</code> structure are as follows:
<i>msg_perm</i>	
<i>uid_t uid</i>	The user ID of the owner of the message queue.
<i>gid_t gid</i>	The group ID of the owner of the message queue.
<i>uid_t cuid</i>	The user ID of the creator of the message queue.
<i>gid_t cgid</i>	The group ID of the creator of the message queue.
<i>mode_t mode</i>	The permissions for the message queue.
<i>msgnum_t</i>	The number of messages currently on the message queue.
<i>msg_qnum</i>	
<i>msglen_t</i>	The maximum number of bytes allowed on the message queue.
<i>msg_qbytes</i>	
<i>pid_t msg_lspid</i>	The process ID of the last job to send a message using <code>msgsnd()</code> .
<i>pid_t msg_lrpid</i>	The process ID of the last job to receive a message using <code>msgrcv()</code> .
<i>time_t msg_stime</i>	The time the last job sent a message to the message queue using <code>msgsnd()</code> .
<i>time_t msg_rtime</i>	The time the last job received a message from the message queue using <code>msgrcv()</code> .
<i>time_t msg_ctime</i>	The time the last job changed the message queue using <code>msgctl()</code> .

## Authorities

### Authorization Required for `msgctl()`

Object Referred to	Authority Required	errno
Message queue for which state information is retrieved ( <i>cmd</i> = <code>IPC_STAT</code> )	Read	EACCES
Message queue for which state information is set ( <i>cmd</i> = <code>IPC_SET</code> )	See Note	EPERM
Message queue to be removed ( <i>cmd</i> = <code>IPC_RMID</code> )	See Note	EPERM

**Note:** To set message queue information or to remove a message queue, the thread must be the owner or creator of the queue, or have appropriate privileges. To increase the maximum number of bytes for the message queue, a thread must have appropriate privileges.

## Return Value

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

## Error Conditions

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

### [EACCES]

Permission denied.

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

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

The *cmd* parameter is `IPC_STAT` and the calling thread does not have read permission.

### [EDAMAGE]

A damaged object was encountered.

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

The message queue has been damaged by a previous message queue operation.

### [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`.

One of the following has occurred:

- The *msgid* parameter is not a valid message queue identifier.
- The *cmd* parameter is not a valid command.
- The *cmd* parameter is `IPC_SET` and the value of *msg\_qbytes* exceeds the system limit.

### [EPERM]

Operation not permitted.

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

The *cmd* parameter is equal to `IPC_RMID` or `IPC_SET` and both of the following are true:

- the caller does not have the appropriate privileges.
- the effective user ID of the caller is not equal to the user ID of the owner or the user ID of the creator of the message queue.

The *cmd* parameter is `IPC_SET` and an attempt is being made to increase the maximum number of bytes for the message queue, but the the caller does not have appropriate privileges.

[EUNKNOWN]

Unknown system state.

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

## Error Messages

None.

## Usage Notes

1. “Appropriate privileges” is defined to be \*ALLOBJ special authority. If the user profile under which the thread is running does not have \*ALLOBJ special authority, the caller does not have appropriate privileges.

## Related Information

- The <sys/msg.h> file (see “Header Files for UNIX-Type Functions” on page 144)
- “Message Queues” on page 136 for the current system limits.
- “msgget()—Get Message Queue”—Get Message Queue
- “msgrcv()—Receive Message Operation” on page 15—Receive Message Operation
- “msgsnd()—Send Message Operation” on page 19—Send Message Operation

## Example

See Code disclaimer information for information pertaining to code examples.

The following example performs a control operation on a message queue:

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

main() {
    int msqid = 2;
    int rc;
    struct msqid_ds buf;

    rc = msgctl(msqid, IPC_STAT, &buf);
}
```

API introduced: V3R6

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

---

## msgget()—Get Message Queue

Syntax

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

int msgget(key_t key, int msgflg);
```

Service Program Name: QP0ZUSHR

Default Public Authority: \*USE

Threadsafe: Yes

The **msgget()** function either creates a new message queue or returns the message queue identifier associated with the *key* parameter for an existing message queue. A new message queue is created if one of the following is true:

- The *key* parameter is equal to `IPC_PRIVATE`.
- The *key* parameter does not already have a message queue identifier associated with it and the `IPC_CREAT` flag is specified in the *msgflg* parameter.

The system maintains status information about a message queue which can be retrieved with the “**msgctl()**—Perform Message Control Operations” on page 8 function. When a new message queue is created, the system initializes the members of the `msgqid_ds` structure as follows:

- `msg_perm.cuid` and `msg_perm.uid` are set equal to the effective user ID of the calling thread.
- `msg_perm.cgid` and `msg_perm.gid` are set equal to the effective group ID of the calling thread.
- The low-order 9 bits of `msg_perm.mode` are set equal to the low-order 9 bits of the *msgflg* parameter.
- `msg_qbytes` is set equal to the system limit.
- `msg_ctime` is set equal to the current time.
- `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` are set equal to 0.

## Parameters

**key** (Input) Key associated with the message queue. A key of `IPC_PRIVATE` (0x00000000) guarantees that a unique message queue is created. A key also can be specified by the caller or generated by the “**ftok()**—Generate IPC Key from File Name” on page 3 function.

### **msgflg**

(Input) Operations and permissions flag. The value of *msgflg* is either zero or is obtained by performing an OR operation on one or more of the following constants:

#### **S\_IRUSR (0x00000100)**

Allow the owner of the message queue to read from it.

#### **S\_IWUSR (0x00000080)**

Allow the owner of the message queue to write to it.

#### **S\_IRGRP (0x00000020)**

Allow the group of the message queue to read from it.

#### **S\_IWGRP (0x00000010)**

Allow the group of the message queue to write to it.

#### **S\_IROTH (0x00000004)**

Allow others to read from the message queue.

#### **S\_IWOTH (0x00000002)**

Allow others to write to the message queue.

#### **IPC\_CREAT (0x00000200)**

Create the message queue if it does not exist.

#### **IPC\_EXCL (0x00000400)**

Return an error if the `IPC_CREAT` flag is set and the message queue already exists.

## Authorities

### Authorization Required for **msgget()**

Object Referred to	Authority Required	errno
Message queue to be created	None	None
Existing message queue to be accessed	See Note	EACCES

**Note:** If the thread is accessing an existing message queue, the mode specified in the last 9 bits of *msgflg* must be a subset of the mode of the existing message queue.

## Return Value

*value*                    **msgget()** was successful. The value returned is the message queue identifier associated with the *key* parameter.

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

## Error Conditions

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

### [EACCES]

Permission denied.

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

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

A message queue identifier exists for the parameter *key*, but permissions specified in the low-order 9 bits of *semflg* are not a subset of the current permissions.

### [EDAMAGE]

A damaged object was encountered.

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

The message queue has been damaged by a previous message queue operation.

### [EEXIST]

File exists.

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

The named file, directory, or path already exists.

A message queue identifier exists for the *key* parameter and both the IPC\_CREAT and IPC\_EXCL flags are set in the *msgflg* parameter.

### [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.

A message queue identifier does not exist for the *key* parameter, and the IPC\_CREAT flag is not set in the *msgflg* parameter.

### [ENOSPC]

No space available.

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

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

A message queue identifier cannot be created because the system limit on the maximum number of allowed message queue identifiers would be exceeded.

[EUNKNOWN]

Unknown system state.

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

## Error Messages

None.

## Usage Notes

1. The best way to generate a unique key is to use the “ftok()—Generate IPC Key from File Name” on page 3 function.

## Related Information

- The <sys/msg.h> file (see “Header Files for UNIX-Type Functions” on page 144)
- “Message Queues” on page 136 for the current system limits.
- “ftok()—Generate IPC Key from File Name” on page 3—Generate IPC Key from File Name
- “msgctl()—Perform Message Control Operations” on page 8—Perform Message Control Operations
- “msgrcv()—Receive Message Operation” on page 15—Receive Message Operation
- “msgsnd()—Send Message Operation” on page 19—Send Message Operation

## Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a message queue:

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

main() {
    int msqid;

    msqid = msgget(IPC_PRIVATE, IPC_CREAT | S_IRUSR | S_IWUSR);
}
```

API introduced: V3R6

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

---

## msgrcv()—Receive Message Operation

```
Syntax
#include <sys/msg.h>

int msgrcv(int msqid, void *msgp, size_t msgsz,
           long int msgtyp, int msgflg);
```

Service Program Name: QP0ZUSHR

Default Public Authority: \*USE

Threadsafe: Yes

The **msgrcv()** function reads a message from the message queue specified by the *msqid* parameter and places it in the user-defined buffer pointed to by the *\*msgp* parameter.

The *\*msgp* parameter points to a user-defined buffer that must contain the following:

1. A field of type `long int` that specifies the type of the message.
2. A data part that contains the data bytes of the message.

The following structure is an example of what this user-defined buffer might look like for a message that has 5 bytes of data:

```
struct mymsg {
    long int    mtype;    /* message type */
    char       mtext[5]; /* message text */
}
```

The value of *mtype* is the type of the received message, as specified by the sender of the message.

The *msgtyp* parameter specifies the type of message to receive from the message queue as follows:

- If *msgtyp* is equal to zero, the first message is received.
- If *msgtyp* is greater than zero, the first message of type *msgtyp* is received.
- If *msgtyp* is less than zero, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

The *msgsz* parameter specifies the size in bytes of the data part of the message. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and the `MSG_NOERROR` flag is set in the *msgflg* parameter. The truncated part of the message is lost and no indication of the truncation is given to the calling thread. Otherwise, **msgrcv()** returns with return value -1 and *errno* set to `E2BIG`.

If a message of the desired type is not available on the message queue, the *msgflg* parameter specifies the action to be taken. The actions are as follows:

- If the `IPC_NOWAIT` flag is set in the *msgflg* parameter, **msgrcv()** returns immediately with a return value of -1 and *errno* set to `ENOMSG`.
- If the `IPC_NOWAIT` flag is not set in the *msgflg* parameter, the calling thread suspends processing until one of the following occurs:
  - A message of the desired type is sent to the message queue.
  - The message queue identifier *msqid* is removed from the system. When this occurs, the **msgrcv()** function returns with a return value of -1 and *errno* set to `EIDRM`.

- A signal is delivered to the calling thread. When this occurs, the **msgrcv()** function returns with a return value of -1 and *errno* set to EINTR.

The system maintains status information about a message queue which can be retrieved with the “msgctl()—Perform Message Control Operations” on page 8 function. When a message is successfully received from the message queue, the system sets the members of the *msqid\_ds* structure as follows:

- *msg\_qnum* is decremented by 1.
- *msg\_lrpid* is set to the process ID of the calling thread.
- *msg\_rtime* is set to the current time.

## Parameters

**msqid** (Input) Message queue identifier, a positive integer. It is returned by the “msgget()—Get Message Queue” on page 11 function and used to identify the message queue to receive the message from.

**msgp** (Output) Pointer to a buffer in which the received message will be stored. See above for the details on the format of the buffer.

**msgsz** (Input) Length of the data part of the buffer.

**msgtyp** (Input) Type of message to be received.

**msgflg** (Input) Operations flags. The value of *msgflg* is either zero or is obtained by performing an OR operation on one or more of the following constants:

**IPC\_NOWAIT (0x00000800)**

If a message is not available, do not wait for the message and return immediately.

**MSG\_NOERROR (0x00001000)**

If the data part of the message is larger than *msgsz*, do not return an error.

## Authorities

### Authorization Required for msgrcv()

Object Referred to	Authority Required	errno
Message queue from which message is received	Read	EACCES

## Return Value

*value*                    **msgrcv()** was successful. The value returned is the number of bytes of data placed in the data part of the buffer pointed to by the *msgp* parameter.

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

## Error Conditions

If **msgrcv()** 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.

The size in bytes of the message is greater than *msgsz* and the MSG\_NOERROR flag is not set in the *msgflg* parameter.

[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.

The calling thread does not have read permission to the message queue.

[EDAMAGE]

A damaged object was encountered.

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

The message queue has been damaged by a previous message queue operation.

[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.

[EIDRM]

ID has been removed.

The message queue identifier *msqid* was removed from the system.

[EINTR]

Interrupted function call.

The function **msgrcv()** was interrupted by a signal.

[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.

One of the following has occurred:

- The *\*msgp* parameter is NULL.
- The *msqid* parameter is not a valid message queue identifier.

[ENOMSG]

Message does not exist.

The message queue does not contain a message of the desired type and the IPC\_NOWAIT flag is set in the *msgflg* parameter.

[EUNKNOWN]

Unknown system state.

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

## Error Messages

None.

## Usage Notes

1. The parameter *msgsz* should include any bytes inserted by the compiler for padding or alignment purposes. These bytes are part of the message data and affect the total number of bytes in the message queue.

The following example shows pad data and how it affects the size of a message using `datamodel *P128`:

```
struct mymsg {
    long int    mtype;        /* 12 bytes padding inserted after */
    char        *pointer;    /* the mtype field by the compiler.*/
    char        c;           /* 15 bytes padding inserted after */
    char        *pointer2;   /* the c field by the compiler.   */
} msg;
/* After the mtype field, there are*/
/* 33 bytes of user data, but 60 */
/* bytes of data including padding.*/
msgsz = sizeof(msg) - sizeof(long int); /* 60 bytes. */
```

2. The `msgrcv()` function does not tag message data with a CCSID (coded character set identifier) value. If a CCSID value is required to correctly interpret the message data, it is the responsibility of the caller to include the CCSID value as part of the data.
3. If the `msgrcv()` function does not complete successfully, the requested message is not removed from the message queue.

## Related Information

- The `<sys/msg.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “Message Queues” on page 136 for the current system limits.
- “msgctl()—Perform Message Control Operations” on page 8—Perform Message Control Operations
- “msgget()—Get Message Queue” on page 11—Get Message Queue
- “msgsnd()—Send Message Operation” on page 19—Send Message Operation
- sigaction()—Examine and change signal action

## Example

See Code disclaimer information for information pertaining to code examples.

The following example receives a message from a message queue:

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

main() {
    int msqid = 2;
    int rc;
    size_t msgsz;
    long int msgtyp;
    struct mymsg {
        long int mtype;
        char    mtext[256];
    };

    msgsz = sizeof(struct mymsg) - sizeof(long int);
    msgtyp = 1;
    rc = msgrcv(msqid, &mymsg, msgsz, msgtyp, IPC_NOWAIT);
}
```

## msgsnd()—Send Message Operation

### Syntax

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

```
int msgsnd(int msqid, void *msgp,
           size_t msgsz, int msgflg);
```

Service Program Name: QP0ZUSHR

Default Public Authority: \*USE

Threadsafe: Yes

The **msgsnd()** function is used to send a message to the message queue specified by the *msqid* parameter.

The *\*msgp* parameter points to a user-defined buffer that must contain the following:

1. A field of type `long int` that specifies the type of the message.
2. A data part that contains the data bytes of the message.

The following structure is an example of what the user-defined buffer might look like for a message that has 5 bytes of data.

```
struct mymsg {
    long int    mtype;    /* message type */
    char       mtext[5]; /* message text */
}
```

The value of *mtype* must be greater than zero. When messages are received with “**msgrcv()**—Receive Message Operation” on page 15, the message type can be used to select the messages. The message data can be any length up to the system limit.

If the message queue is full, the *msgflg* parameter specifies the action to be taken. The actions are as follows:

- If the `IPC_NOWAIT` flag is set in the *msgflg* parameter, the message is not sent. The **msgsnd()** function returns immediately with a return value of -1 and *errno* set to `EAGAIN`.
- If the `IPC_NOWAIT` flag is not set in the *msgflg* parameter, the calling thread suspends processing until one of the following occurs:
  - Enough messages are received from the message queue so that a message of size *msgsz* can be placed on the message queue.
  - The message queue identifier *msqid* is removed from the system. When this occurs, the **msgsnd()** function returns with a return value of -1 and *errno* set to `EIDRM`.
  - A signal is delivered to the calling thread. When this occurs, the **msgsnd()** function returns with a return value of -1 and *errno* set to `EINTR`.

The system maintains status information about a message queue which can be retrieved with the “msgctl()—Perform Message Control Operations” on page 8 function. When a message is successfully sent to the message queue, the system sets the members of the `msqid_ds` structure as follows:

- `msg_qnum` is incremented by 1.
- `msg_lspid` is set to the process ID of the calling thread.
- `msg_stime` is set to the current time.

## Parameters

**msqid** (Input) Message queue identifier, a positive integer. It is returned by the “msgget()—Get Message Queue” on page 11 function and used to identify the message queue to send the message to.

**msgp** (Input) Pointer to a buffer with the message to be sent. See above for the details on the format of the buffer.

**msgsz** (Input) Length of the data part of the message to be sent.

**msgflg**

(Input) Operations flags. The value of `msgflg` is either zero or is obtained by performing an OR operation on one or more of the following constants:

**IPC\_NOWAIT (0x00000800)**

If the message queue is full, do not wait for space to become available on the message queue and return immediately.

## Authorities

Authorization Required for `msgsnd()`

Object Referred to	Authority Required	errno
Message queue on which message is placed	Write	EACCES

## Return Value

0 `msgsnd()` was successful.

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

## Error Conditions

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

[EACCES]

Permission denied.

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

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

The calling thread does not have write permission to the message queue.

[EAGAIN]

Operation would have caused the process to be suspended.

The message cannot be sent for one of the reasons cited above and the `IPC_NOWAIT` flag is set in the `msgflg` parameter.

#### [EDAMAGE]

A damaged object was encountered.

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

The message queue has been damaged by a previous message queue operation.

#### [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.

#### [EIDRM]

ID has been removed.

The message queue identifier `msqid` was removed from the system.

#### [EINTR]

Interrupted function call.

The function `msgsnd()` was interrupted by a signal.

#### [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.

One of the following has occurred:

- The `*msgp` parameter is NULL.
- The `msqid` parameter is not a valid message queue identifier.
- The `mtype` parameter is less than or equal to zero.
- The `msgsz` parameter is greater than the system limit.

#### [EUNKNOWN]

Unknown system state.

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

## Error Messages

None.

## Usage Notes

1. The parameter `msgsz` should include any bytes inserted by the compiler for padding or alignment purposes. These bytes are part of the message data and affect the total number of bytes in the message queue.

The following example shows pad data and how it affects the size of a message when using `datamodel *P128`:

```

struct mymsg {
    long int    mtype;    /* 12 bytes padding inserted after */
    char        *pointer; /* the mtype field by the compiler.*/
    char        c;       /* 15 bytes padding inserted after */
    char        *pointer2; /* the c field by the compiler. */
} msg;
/* After the mtype field, there are*/
/* 33 bytes of user data, but 60 */
/* bytes of data including padding.*/
msgsz = sizeof(msg) - sizeof(long int);    /* 60 bytes. */

```

2. The `msgsnd()` function does not tag message data with a CCSID (coded character set identifier) value. If a CCSID value is required to correctly interpret the message data, it is the responsibility of the caller to include the CCSID value as part of the data.
3. If the `msgsnd()` function does not complete successfully, the requested message is not placed on the message queue.

## Related Information

- The `<sys/msg.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “Message Queues” on page 136 for the current system limits.
- “msgctl()—Perform Message Control Operations” on page 8—Perform Message Control Operations
- “msgget()—Get Message Queue” on page 11—Get Message Queue
- “msgrcv()—Receive Message Operation” on page 15—Receive Message Operation
- `sigaction()`—Examine and change signal action

## Example

See Code disclaimer information for information pertaining to code examples.

The following example sends a message to a message queue:

```

#include <sys/msg.h>

main() {
    int msqid = 2;
    int rc;
    size_t msgsz;
    struct mymsg {
        long int mtype;
        char    mtext[256];
    };

    msgsz = sizeof(struct mymsg) - sizeof(long int);
    mymsg.mtype = 1;
    rc = msgsnd(msqid, &mymsg, msgsz, IPC_NOWAIT);
}

```

API introduced: V3R6

Top | UNIX-Type APIs | APIs by category

---

## QlgFtok()—Generate IPC Key from File Name (using NLS-enabled path name)

### Syntax

```
#include <sys/ipc.h>
#include <qlg.h>
```

```
key_t QlgFtok(const Qlg_Path_Name_T *path, int id);
```

Service Program Name: QP0ZCPA

Default Public Authority: \*USE

Threadsafe: Conditional. See Usage Notes for “ftok()—Generate IPC Key from File Name” on page 3.

The **QlgFtok()** function, like the **ftok()** function, generates an IPC key based on the combination of *path* and *id*. The difference is that the **QlgFtok()** function takes a pointer to a `Qlg_Path_Name_T` structure, while the **ftok()** function takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, returnvalues, and related information, see “ftok()—Generate IPC Key from File Name” on page 3.

## Parameters

**path** (Input) The path name of the file used in combination with *id* to generate the key. For more information on the `Qlg_Path_Name_T` structure, see Path name format.

## Related Information

“ftok()—Generate IPC Key from File Name” on page 3—Generate IPC Key from File Name

## Example

See Code disclaimer information for information pertaining to code examples.

The following example uses the **QlgFtok()** and **semget()** functions.

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>
#include <stdio.h>
#include <qlg.h>

int main(int argc, char *argv[])
{
    key_t myKey;
    int  semid;

    #define mypath "/myApplication/myFile"
    const char US_const[3]= "US";
    const char Language_const[4]="ENU";
    const char Path_Name_Del_const[2]= "/";
    typedef struct pnstruct
    {
        Qlg_Path_Name_T qlg_struct;
        char[100] pn; /* This size must be >= the path */
                    /* name length or be a pointer */
                    /* to the path name. */
    }
```

```

    };
    struct pnstruct path;

    /*****
    /* Initialize Qlg_Path_Name_T parameters */
    /*****
memset((void*)path name, 0x00, sizeof(struct pnstruct));
path.qlg_struct.CCSID = 37;
memcpy(path.qlg_struct.Country_ID,US_const,2);
memcpy(path.qlg_struct.Language_ID,Language_const,3);
path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
path.qlg_struct.Path_Length = sizeof(mypath)-1;
memcpy(path.qlg_struct.Path_Name_Delimiter,Path_Name_De1_const,1);
memcpy(path.pn,mypath,sizeof(mypath));

/* Use QlgFtok to generate a key associated with a file. */
/* Every process will get the same key back if the caller */
/* calls with the same parameters. */
myKey = QlgFtok((Qlg_Path_Name_T *)path name, 42);
if(myKey == -1) {
    printf("QlgFtok failed with errno = %d\n", errno);
    return -1;
}

/* Call an xxxget() API, where xxx is sem, shm, or msg. */
/* This will create or reference an existing IPC object */
/* with the 'well known' key associated with the file */
/* name used above. */
semid = semget(myKey, 1, 0666 | IPC_CREAT);
if(semid == -1) {
    printf("semget failed with errno = %d\n", errno);
    return -1;
}

/* ... Use the semaphore as required ... */
return 0;
}

```

API introduced: V5R1

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

---

## QlgSem\_open()—Open Named Semaphore (using NLS-enabled path name)

### Syntax

```

#include <semaphore.h>
#include <qlg.h>

sem_t * QlgSem_open(const Qlg_Path_Name_T *name,
                   int oflag, ...);

```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes



The `QlgSem_open()` function, like the `sem_open()` function, opens a named semaphore and returns a semaphore pointer that may be used on subsequent calls to `sem_post()`, `sem_post_np()`, `sem_wait()`, `sem_wait_np()`, `sem_trywait()`, `sem_getvalue()`, and `sem_close()`. The `QlgSem_open()` function takes a pointer to a `Qlg_Path_Name_T` structure, while the `sem_open()` function takes a pointer to a character string that is in the CCSID of the job.

Limited information on the *name* parameter is provided in this API. For additional information on the *name* parameter and a discussion of other parameters, authorities required, return values, and related information, see “`sem_open()`—Open Named Semaphore” on page 78—Open Named Semaphore.

## Parameters

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

## Error Conditions

If `QlgSem_open()` is not successful, *errno* usually indicates the following error or one of the errors identified in “Error Conditions” on page 79—Open Named Semaphore.

[ECONVERT]

A conversion error for the parameter *name*.

## Related Information

- The `<qlg.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`sem_open()`—Open Named Semaphore” on page 78—Open Named Semaphore
- “`QlgSem_open_np()`—Open Named Semaphore with Maximum Value (using NLS-enabled path name)” on page 26—Open Named Semaphore with Maximum Value (using NLS-enabled path name)
- “`QlgSem_unlink()`—Unlink Named Semaphore (using NLS-enabled path name)” on page 28—Unlink Named Semaphore (using NLS-enabled path name)

**Note:** All of the related information for `sem_open()` applies to `QlgSem_open()`. See Related Information in “Related Information” on page 80.

## Example

See Code disclaimer information for information pertaining to code examples.

The following example opens the named semaphore “/mysemaphore” and creates the semaphore with an initial value of 10 if it does not already exist. If the semaphore is created, the permissions are set such that only the current user has access to the semaphore.

```
#include <semaphore.h>
#include <qlg.h>
main() {

    sem_t * my_semaphore;
    int rc;

    #define mypath "/mysemaphore"
    const char US_const[3]= "US";
    const char Language_const[4]="ENU";
    const char Path_Name_De]_const[2]= "/";
    typedef struct pnstruct
    {
        Qlg_Path_Name_T qlg_struct;
        char[100] pn; /* This size must be >= the path */
                    /* name length or be a pointer */
                    /* to the path name. */
    }
```

```

};
struct pnstruct path;

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

my_semaphore = QlgSem_open((Qlg_Path_Name_T *)path.name,
                          O_CREAT, S_IRUSR | S_IWUSR, 10);
}

```

API introduced: V5R1

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

---

## QlgSem\_open\_np()—Open Named Semaphore with Maximum Value (using NLS-enabled path name)

```

Syntax
#include <semaphore.h>
#include <qlg.h>

sem_t * QlgSem_open_np(const Qlg_Path_Name_T *name, int oflag,
                      mode_t mode, unsigned int value,
                      sem_attr_np_t *attr);

```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes

The **QlgSem\_open\_np()** function, like the **sem\_open\_np()** function, opens a named semaphore and returns a semaphore pointer that may be used on subsequent calls to **sem\_post()**, **sem\_post\_np()**, **sem\_wait()**, **sem\_wait\_np()**, **sem\_trywait()**, **sem\_getvalue()**, and **sem\_close()**. The **QlgSem\_open\_np()** function takes a pointer to a **Qlg\_Path\_Name\_T** structure, while the **sem\_open\_np()** function takes a pointer to a character string.

Limited information on the *name* parameter is provided in this API. For additional information on the *name* parameter and a discussion of other parameters, authorities required, return values, and related information, see “**sem\_open\_np()—Open Named Semaphore with Maximum Value**” on page 81—Open Named Semaphore with Maximum Value.

## Parameters

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

## Error Conditions

If `QlgSem_open_np()` is not successful, *errno* usually indicates the following error or one of the errors identified in “Error Conditions” on page 83—Open Named Semaphore with Maximum Value.

[ECONVERT]

A conversion error for the parameter *name*.

## Related Information

- The `<qlg.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`sem_open_np()`—Open Named Semaphore with Maximum Value” on page 81—Open Named Semaphore with Maximum Value
- “`QlgSem_open()`—Open Named Semaphore (using NLS-enabled path name)” on page 24—Open Named Semaphore (using NLS-enabled path name)
- “`QlgSem_unlink()`—Unlink Named Semaphore (using NLS-enabled path name)” on page 28—Unlink Named Semaphore (using NLS-enabled path name)

**Note:** All of the related information for `sem_open_np()` applies to `QlgSem_open_np()`. See Related Information in “Related Information” on page 84.

## Example

See Code disclaimer information for information pertaining to code examples.

The following example opens the named semaphore `"/mysemaphore"` and creates the semaphore with an initial value of 10 and a maximum value of 11. The permissions are set such that only the current user has access to the semaphore.

```
#include <semaphore.h>
#include <qlg.h>
main() {

    sem_t * my_semaphore;
    int rc;
    sem_attr_np_t attr;

    #define mypath "/mysemaphore"
    const char US_const[3]= "US";
    const char Language_const[4]="ENU";
    const char Path_Name_Del_const[2]= "/";
    typedef struct pnstruct
    {
        Qlg_Path_Name_T qlg_struct;
        char[100] pn; /* This size must be >= the path */
                        /* name length or be a pointer */
                        /* to the path name. */
    };
    struct pnstruct path;

    /******
    /* Initialize Qlg_Path_Name_T parameters */
    /******
    memset((void*)path.name, 0x00, sizeof(struct pnstruct));
    path.qlg_struct.CCSID = 37;
```

```

memcpy(path.qlg_struct.Country_ID,US_const,2);
memcpy(path.qlg_struct.Language_ID,Language_const,3);
path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
path.qlg_struct.Path_Length = sizeof(mypath)-1;
memcpy(path.qlg_struct.Path_Name_Delimiter,Path_Name_Del_const,1);
memcpy(path.pn,mypath,sizeof(mypath));

memset(&attr, 0, sizeof(attr));
attr.maxvalue=11;
my_semaphore = QlgSem_open_np((Qlg_Path_Name_T *)path name,
                             O_CREAT|O_EXCL,
                             S_IRUSR | S_IWUSR,
                             10,
                             &attr);
}

```

API introduced: V5R1

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

---

## QlgSem\_unlink()—Unlink Named Semaphore (using NLS-enabled path name)

### Syntax

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

```
int QlgSem_unlink(const Qlg_Path_Name_T *name);
```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes

The **QlgSem\_unlink()** function, like the **sem\_unlink()** function, unlinks a named semaphore. The **QlgSem\_unlink()** function takes a pointer to a `Qlg_Path_Name_T` structure, while the **sem\_unlink()** function takes a pointer to a character string.

Limited information on the *name* parameter is provided in this API. For additional information on the *name* parameter, authorities required, return values, and related information, see “[sem\\_unlink\(\)—Unlink Named Semaphore](#)” on page 91—Unlink Named Semaphore.

## Parameters

**name** (Input) A pointer a `Qlg_Path_Name_T` structure that contains a path name or a pointer to a path name of the semaphore to be unlinked. For more information on the `Qlg_Path_Name_T` structure, see [Path name format](#).

## Error Conditions

If **QlgSem\_unlink()** is not successful, *errno* usually indicates the following error or one of the errors identified in “[sem\\_unlink\(\)—Unlink Named Semaphore](#)” on page 91—Unlink Named Semaphore.

[ECONVERT]

A conversion error for the parameter *name*.

## Related Information

- The `<qlg.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`sem_unlink()`—Unlink Named Semaphore” on page 91—Unlink Named Semaphore
- “`QlgSem_open()`—Open Named Semaphore (using NLS-enabled path name)” on page 24—Open Named Semaphore (using NLS-enabled path name)
- “`QlgSem_open_np()`—Open Named Semaphore with Maximum Value (using NLS-enabled path name)” on page 26—Open Named Semaphore with Maximum Value (using NLS-enabled path name)

**Note:** All of the related information for `sem_unlink()` applies to `QlgSem_unlink()`. See Related Information in “Related Information” on page 92.

## Example

See Code disclaimer information for information pertaining to code examples.

The following example unlinks the named semaphore `"/mysem"`.

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

main() {
    int rc;

    #define mypath "/mysem"
    const char US_const[3]= "US";
    const char Language_const[4]="ENU";
    const char Path_Name_Del_const[2]= "/";
    typedef struct pnstruct
    {
        Qlg_Path_Name_T qlg_struct;
        char[100] pn; /* This size must be >= the path */
                    /* name length or be a pointer */
                    /* to the path name. */

    };
    struct pnstruct path;

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

    rc = QlgSem_unlink((Qlg_Path_Name_T *)path.name);
}

```

API introduced: V5R1

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

## Delete Interprocess Communication Objects (QP0ZDIPC) API

Required Parameter Group:	
1	Delete control
<b>Input</b>	Char(*)
2	Error code
<b>I/O</b>	Char(*)
Default Public Authority: *USE	
Threadsafe: No	

The Delete Interprocess Communication Objects (QP0ZDIPC) API deletes one or more interprocess communication (IPC) objects as specified by the delete control parameter.

### Authorities and Locks

#### *Job Authority*

The calling thread must be the owner, must be the creator, or must have \*ALLOBJ special authority.

For additional information on these authorities, see the iSeries Security Reference  book.

### Required Parameter Group

#### Delete control

INPUT; CHAR(\*)

Information about which IPC objects to delete. For the layout of this structure, see "Delete Control Format."

#### Error code

I/O; CHAR(\*)

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

### Delete Control Format

The following shows the format of the delete control parameter. For detailed descriptions of the fields in the table, see "Field Descriptions" on page 31.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Number of objects to delete.
These fields repeat for each object to delete.		CHAR(1)	IPC type
		CHAR(3)	Reserved
		BINARY(4)	Identifier

## Field Descriptions

**Identifier.** A unique IPC identifier that is used to specify which IPC object is to be deleted. The identifier is obtained from calling the APIs `semget()`, `shmget()`, `msgget()`, or `QP0ZOLIP`.

**IPC type.** This value describes the type of IPC object to delete. Possible values follow:

- 1 Delete a semaphore set object.
- 2 Delete a shared memory object.
- 3 Delete a message queue object.

**Number of objects to delete.** The number of IPC objects in the delete control parameter.

**Reserved.** A reserved field. These characters must be set to '00'x.

## Error Messages

Message ID	Error Message Text
CPF24B4 E	Severe error while addressing parameter list.
CPF3C90 E	Literal value cannot be changed.
CPF3CF1 E	Error code parameter not valid.
CPFA986 E	&1 IPC objects deleted; &2 IPC object not deleted
CPDA981 D	Not authorized to delete IPC object &1.
CPDA982 D	IPC object &1 does not exist.
CPDA983 D	IPC object &1 is marked as damaged.
CPFA987 E	Delete control not valid.

API introduced: V4R2

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

## Open List of Interprocess Communication Objects (QP0ZOLIP) API

Required Parameter Group:	
1	Receiver variable
<b>Output</b>	Char(*)
2	Length of receiver variable
<b>Input</b>	Binary(4)
3	List information
<b>Output</b>	Char(80)
4	Number of records to return
<b>Input</b>	Binary(4)
5	Format name
<b>Input</b>	Char(8)
6	Filter information
<b>Input</b>	Char(*)
7	Filter format name
<b>Input</b>	Char(8)
8	Error code
<b>I/O</b>	Char(*)
Default Public Authority: *USE	
Threadsafe: No	

The Open List of Interprocess Communication Objects (QP0ZOLIP) API lets you generate a list of interprocess communication (IPC) objects and descriptive information based on the selection parameters. The QP0ZOLIP API places the specified number of list entries in the receiver variable. You can access additional records by using the Get List Entries (QGYGTLE) API. On successful completion of the QP0ZOLIP API, a handle is returned in the list information parameter. You may use this handle on subsequent calls to the following APIs:

- Get List Entries (QGYGTLE)
- Find Entry Number in List (QGYFNDE)
- Close List (QGYCLST)

You can use the QP0ZOLIP API to:

- Open a list of all IPC objects of a specific type (semaphore sets, message queues, shared memory, named semaphores, or unnamed semaphores).
- Open a list of identifier-based IPC objects (semaphore sets, message queues, or shared memory) of a specific type with a key in a specified range.
- Open a list of identifier-based IPC objects of a specific type that are owned by one or more specified users.
- Open a list of IPC objects of a specific type (semaphore sets, message queues, shared memory, or named semaphores) that were created by one or more specified users.




Only one IPC type (either semaphore sets, message queue, shared memory, named semaphores, or unnamed semaphores) can be returned in one call to this API. The IPC type is determined by the format name parameter.

The records returned by QP0ZOLIP include an information status field that describes the completeness and validity of the information. Be sure to check the information status field before using any other information returned.

## Authorities and Locks

### *Job Authority*

Service special authority (\*SERVICE) is needed to call this API.

For additional information on this authority, see the iSeries Security Reference  book.

## Required Parameter Group

### Receiver variable

OUTPUT; CHAR(\*)

The variable that is used to return the IPC object information that you requested.

### Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable.

### List information

OUTPUT; CHAR(80)

Information about the list of IPC objects that were opened. For a description of the layout of this parameter, see Format of Open List Information.

### Number of records to return

INPUT; BINARY(4)

The number of records in the list to put into the receiver variable.

### Format name

INPUT; CHAR(8)

The format of the information to be returned in the receiver variable. This parameter will determine the type of IPC mechanism to open the list for. You must use one of the following format names:

<i>LSST0100</i>	This format is described in “LSST0100 Format” on page 35.
<i>LMSQ0100</i>	This format is described in “LMSQ0100 Format” on page 36.
<i>LSHM0100</i>	This format is described in “LSHM0100 Format” on page 36.
<i>LNSM0100</i>	This format is described in “LNSM0100 Format” on page 37.
<i>LUSM0100</i>	This format is described in “LUSM0100 Format” on page 38.

### Filter information

INPUT; CHAR(\*)

The information in this parameter is used to filter the list of IPC objects. The format of this variable depends on the filter format name.

### Filter format name

INPUT; CHAR(8)

The name of the format that is used to filter the list of IPC objects. You must use one of the following format names:

FIPC0100 This format is described in "FIPC0100 Format."

**Error code**

I/O; CHAR(\*)

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

**FIPC0100 Format**

The following shows the format of the filter information for the FIPC0100 format. For detailed descriptions of the field in the table, see "Field Descriptions."

Offset		Type	Field
Dec	Hex		
0	0	CHAR(1)	Filter on key
1	1	CHAR(3)	Reserved
4	4	BINARY(4)	Minimum key
8	8	BINARY(4)	Maximum key
12	C	BINARY(4)	Offset to owner profiles array
16	10	BINARY(4)	Number of owner profiles specified
20	14	BINARY(4)	Offset to creator profiles array
24	18	BINARY(4)	Number of creator profiles specified
This field repeats for each owner profile name.		CHAR(10)	Owner profile name
This field repeats for each creator profile name.		CHAR(10)	Creator profile name

**Field Descriptions**

**Creator profile name.** The user profile names that created the IPC objects being returned. These values are used only if the number of creator profiles specified field is greater than one. Possible special values follow:

- \*ALL IPC objects created by any user profile are added to the list. The rest of the user profiles in the array are ignored.
- \*CURRENT IPC objects created by the current user profile are added to the list.

**Filter on key.** Whether filtering will be done based on the key value of the IPC object. Possible values follow:

- 0 No filtering is done based on the key value. The values of minimum key field and maximum key field are ignored.
- 1 Filtering is done based on the values of minimum key field and maximum key field.

**Maximum key.** The maximum IPC object's key value. Only the IPC objects with a key greater than or equal to the minimum key and less than or equal to the maximum key will be added to the generated list. This value is only used if the filter on key field is set to one.

**Minimum key.** The minimum IPC object's key value. Only the IPC objects with a key greater than or equal to the minimum key and less than or equal to the maximum key will be added to the generated list. This value is only used if the filter on key field is set to one.

**Number of creator profiles specified.** The number of creator profiles specified in the creator profile names array. If this value is zero, no filtering is to be done for the creator user profile.

**Number of owner profiles specified.** The number of owner profiles specified in the owner profile names array. If this value is zero, no filtering is to be done for the owner user profile.

**Offset to creator profiles array.** The offset in characters (bytes) from the beginning of the filter information to the beginning of the array of creator profiles.

**Offset to owner profiles array.** The offset in characters (bytes) from the beginning of the filter information to the beginning of the array of owner profiles.

**Owner profile name.** The user profile names that own the IPC objects being returned. These values are used only if the number of owner profiles specified field is greater than one. Possible special values follow:

- \*ALL           IPC objects that are owned by any user profile are added to the list. The rest of the user profiles in the array are ignored.
- \*CURRENT     IPC objects that are owned by the current user profile are added to the list.

**Reserved.**These characters must be set to '00'x.

## LSST0100 Format

This format name is used to return list information for semaphore sets. The following table shows the information returned in each record in the receiver variable for the LSST0100 format. For a detailed description of each field, see "Field Descriptions" on page 39.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Identifier
4	4	BINARY(4)	Key
8	8	BINARY(4)	Number of semaphores
12	C	CHAR(1)	Damaged
13	D	CHAR(1)	Owner read permission
14	E	CHAR(1)	Owner write permission
15	F	CHAR(1)	Group read permission
16	10	CHAR(1)	Group write permission
17	11	CHAR(1)	General read permission
18	12	CHAR(1)	General write permission
19	13	CHAR(1)	Authorized to delete
20	14	CHAR(16)	Last semop() date and time
36	24	CHAR(16)	Last administration change date and time
52	34	CHAR(10)	Owner
62	3E	CHAR(10)	Group owner
72	48	CHAR(10)	Creator

Offset		Type	Field
Dec	Hex		
82	52	CHAR(10)	Creator's group

## LMSQ0100 Format

This format name is used to return list information for message queues. The following table shows the information returned in each record in the receiver variable for the LMSQ0100 format. For a detailed description of each field, see "Field Descriptions" on page 39.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Identifier
4	4	BINARY(4)	Key
8	8	CHAR(1)	Damaged
9	9	CHAR(1)	Owner read permission
10	A	CHAR(1)	Owner write permission
11	B	CHAR(1)	Group read permission
12	C	CHAR(1)	Group write permission
13	D	CHAR(1)	General read permission
14	E	CHAR(1)	General write permission
15	F	CHAR(1)	Authorized to delete
16	10	BINARY(4)	Number of messages on queue
20	14	BINARY(4)	Size of all messages on queue
24	18	BINARY(4)	Maximum size of all messages on queue
28	1C	BINARY(4)	Number of threads to receive message
32	20	BINARY(4)	Number of threads to send message
36	24	CHAR(16)	Last msgrcv() date and time
52	34	CHAR(16)	Last msgsnd() date and time
68	44	CHAR(16)	Last administration change date and time
84	54	CHAR(10)	Owner
94	5E	CHAR(10)	Group owner
104	68	CHAR(10)	Creator
114	72	CHAR(10)	Creator's group

## LSHM0100 Format

This format name is used to return list information for shared memory. The following table shows the information returned in each record in the receiver variable for the LSHM0100 format. For a detailed description of each field, see "Field Descriptions" on page 39.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Identifier

Offset		Type	Field
Dec	Hex		
4	4	BINARY(4)	Key
8	8	CHAR(1)	Damaged
9	9	CHAR(1)	Owner read permission
10	A	CHAR(1)	Owner write permission
11	B	CHAR(1)	Group read permission
12	C	CHAR(1)	Group write permission
13	D	CHAR(1)	General read permission
14	E	CHAR(1)	General write permission
15	F	CHAR(1)	Marked to be deleted
16	10	CHAR(1)	Authorized to delete
17	11	CHAR(1)	Teraspace
18	12	CHAR(1)	Resize
19	13	CHAR(1)	Reserved
20	14	BINARY(4)	Segment size
24	18	BINARY(4)	Number attached
28	1C	CHAR(16)	Last shmat() date and time
44	2C	CHAR(16)	Last detach date and time
60	3C	CHAR(16)	Last administration change date and time
76	4C	CHAR(10)	Owner
86	56	CHAR(10)	Group owner
96	60	CHAR(10)	Creator
106	6A	CHAR(10)	Creator's group

## LNSM0100 Format

This format name is used to return list information for named semaphores. The following table shows the information returned in each record in the receiver variable for the LNSM0100 format. For a detailed description of each field, see "Field Descriptions" on page 39.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Length of entry
4	4	BINARY(4)	Value
8	8	BINARY(4)	Maximum value
12	C	BINARY(4)	Offset to waiting threads
16	10	BINARY(4)	Number of waiting threads
20	14	BINARY(4)	Offset to name
24	18	BINARY(4)	Length of name
28	1C	CHAR(16)	Title
44	2C	CHAR(1)	Marked to be deleted
45	2D	CHAR(1)	Authorized to delete

Offset		Type	Field
Dec	Hex		
46	2E	CHAR(10)	Creator
56	38	CHAR(10)	Creator's group
66	42	CHAR(1)	Owner read permission
67	43	CHAR(1)	Owner write permission
68	44	CHAR(1)	Group read permission
69	45	CHAR(1)	Group write permission
70	46	CHAR(1)	General read permission
71	47	CHAR(1)	General write permission
72	48	CHAR(26)	Last sem_post() qualified job identifier
98	62	CHAR(2)	Reserved
100	64	CHAR(16)	Last sem_post() thread identifier
116	74	CHAR(26)	Last sem_wait() qualified job identifier
142	8e	CHAR(2)	Reserved
144	90	CHAR(16)	Last sem_wait() thread identifier
These fields repeat for each thread waiting on the semaphore.		CHAR(26)	Waiting qualified job identifier
		CHAR(2)	Reserved
		CHAR(16)	Waiting thread identifier
This field follows the list of threads waiting on the semaphore.		CHAR(*)	Name of the semaphore

## LUSM0100 Format

This format name is used to return list information for unnamed semaphores. The following table shows the information returned in each record in the receiver variable for the LUSM0100 format. For a detailed description of each field, see "Field Descriptions" on page 39.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Length of entry
4	4	BINARY(4)	Value
8	8	BINARY(4)	Maximum value
12	C	BINARY(4)	Offset to waiting threads
16	10	BINARY(4)	Number of waiting threads
20	14	BINARY(4)	Reserved
24	18	CHAR(16)	Title
40	28	CHAR(26)	Last sem_post() qualified job identifier
66	42	CHAR(2)	Reserved
68	44	CHAR(16)	Last sem_post() thread identifier
84	54	CHAR(26)	Last sem_wait() qualified job identifier
110	6E	CHAR(2)	Reserved

Offset		Type	Field
Dec	Hex		
112	70	CHAR(16)	Last sem_wait() thread identifier
These fields repeat for each thread waiting on the semaphore.		CHAR(26)	Waiting qualified job identifier
		CHAR(2)	Reserved
		CHAR(16)	Waiting thread identifier

## Field Descriptions

**Authorized to delete.** This value determines if the caller has the authority to delete this IPC object. Possible values follow:

- 0 The calling thread cannot delete the IPC object.
- 1 The calling thread can delete the IPC object.

**Creator.** The name of the user profile that created this IPC object.

**Creator's group.** The name of the group profile that created this IPC object. A special value can be returned:

- \*NONE The creator does not have a group profile.

**Damaged.** Whether the IPC object has suffered internal damage. Possible values follow:

- 0 The IPC object is not damaged.
- 1 The IPC object is damaged.

**General read permission.** Whether any user other than the owner and group owner has read authority to the IPC object. Possible values follow:

- 0 General read authority is not allowed to the IPC object.
- 1 General read authority is allowed for the IPC object.

**General write permission.** Whether if any user other than the owner and group owner has write authority to the IPC object. Possible values follow:

- 0 General write authority is not allowed to the IPC object.
- 1 General write authority is allowed to the IPC object.

**Group owner.** The name of the group profile that owns this IPC object. A special value can be returned:

- \*NONE The IPC object does not have a group owner.

**Group read permission.** Whether the group owner has read authority to the IPC object. Possible values follow:

- 0 The group owner does not have read authority to the IPC object.
- 1 The group owner has read authority to the IPC object.

**Group write permission.** Whether the group owner has write authority to the IPC object. Possible values follow:

- 0 The group owner does not have write authority to the IPC object.
- 1 The group owner has write authority to the IPC object.

**Identifier.** The unique IPC object identifier.

**Key.** The key of the IPC object. If this value is zero, this IPC object has no key associated with it.

**Last administration change date and time.** The date and time of the last change to the IPC object for the owner, group owner, or permissions. The 16 characters are:

- 1 Century, where 0 indicates years 19xx and 1 indicates years 20xx.
- 2-7 Date, in YYMMDD (year, month, and day) format.
- 8-13 Time of day, in HHMMSS (hours, minutes, and seconds) format.
- 14-16 Milliseconds.

**Last detach date and time.** The date and time of the last detachment from the shared memory segment. If no thread has performed a successful detachment, this value will be set to all zeros. The 16 characters are:

- 1 Century, where 0 indicates years 19xx and 1 indicates years 20xx.
- 2-7 Date, in YYMMDD (year, month, and day) format.
- 8-13 Time of day, in HHMMSS (hours, minutes, and seconds) format.
- 14-16 Milliseconds.

**Last msgrcv() date and time.** The date and time of the last successful msgrcv() call. If no thread has performed a successful msgrcv() call, this value will be set to all zeros. The 16 characters are:

- 1 Century, where 0 indicates years 19xx and 1 indicates years 20xx.
- 2-7 Date, in YYMMDD (year, month, and day) format.
- 8-13 Time of day, in HHMMSS (hours, minutes, and seconds) format.
- 14-16 Milliseconds.

**Last msgsnd() date and time.** The date and time of the last successful msgsnd() call. If no thread has performed a successful msgsnd() call, this value will be set to all zeros. The 16 characters are:

- 1 Century, where 0 indicates years 19xx and 1 indicates years 20xx.
- 2-7 Date, in YYMMDD (year, month, and day) format.
- 8-13 Time of day, in HHMMSS (hours, minutes, and seconds) format.
- 14-16 Milliseconds.

**Last sem\_post() qualified job identifier.** The job name, the job user profile, and the job number of the last thread that successfully called sem\_post() or sem\_post\_np() if the job has not ended. The 26 characters are:

- 1-10 The job name
- 11-20 The user profile
- 21-26 The job number



If the thread has ended, then the first 16 characters contain 16 characters that uniquely identify the ended job, followed by 10 spaces. If no thread has used `sem_post()` to post to the semaphore, then the 26 characters will contain spaces.

**Last `sem_post()` thread identifier.** The thread ID of the last thread that successfully called `sem_post()` or `sem_post_np()` if the thread has not ended.

**Last `sem_wait()` qualified job identifier.** The job name, the job user profile, and the job number of the last thread that returned from a `sem_wait()`, `sem_wait_np()`, or `sem_wait()` call, if the job has not ended. The 26 characters are:

1-10	The job name
11-20	The user profile
21-26	The job number

If the thread has ended, then the first 16 characters contain 16 characters that uniquely identify the ended job, followed by 10 spaces. If no job has used `sem_wait()` to wait on the semaphore, then the 26 characters will contain spaces.

**Last `sem_wait()` thread identifier.** The thread ID of the last thread that returned from a `sem_wait()`, `sem_wait_np()`, or `sem_wait()` call, if the thread has not ended.

**Last `semop()` date and time.** The date and time of the last successful `semop()` call. If no thread has performed a successful `semop()` call, this value will be set to all zeros. The 16 characters are:

1	Century, where 0 indicates years 19xx and 1 indicates years 20xx.
2-7	Date, in YYMMDD (year, month, and day) format.
8-13	Time of day, in HHMMSS (hours, minutes, and seconds) format.
14-16	Milliseconds.

**Last `shmat()` date and time.** The date and time of the last successful `shmat()`. If no thread has performed a successful `shmat()` call, this value will be set to all zeros. The 16 characters are:

1	Century, where 0 indicates years 19xx and 1 indicates years 20xx.
2-7	Date, in YYMMDD (year, month, and day) format.
8-13	Time of day, in HHMMSS (hours, minutes, and seconds) format.
14-16	Milliseconds.

**Length of entry.** The length of this record in the list.

**Length of name.** The number of bytes in the name of the semaphore, not including the terminating null character.

**Marked to be deleted.** Whether the shared memory is marked to be deleted when the number attached becomes zero. Possible values follow:

0	The shared memory segment is not marked for deletion.
1	The shared memory segment is marked for deletion.

**Maximum size of all messages on queue.** The maximum byte size of all messages that can be on the queue at one time.

**Maximum value.** The maximum value of the semaphore.

**Name of the semaphore.** The null-terminated name of the semaphore.

**Number attached.** The number of times any thread has done a `shmat()` without doing a `detach`.

**Number of messages on queue.** The number of messages that are currently on the message queue.

**Number of semaphores.** The number of semaphores in the semaphore set.

**Number of threads to receive message.** The number of threads that are currently waiting to receive a message.

**Number of threads to send message.** The number of threads that are currently waiting to send a message.

**Number of waiting threads.** The total number of threads that are waiting for this semaphore to reach a certain value.

**Offset to name.** The offset to where the name field begins.

**Offset to waiting threads.** The offset to where the fields containing waiting threads begin.

**Owner.** The name of the user profile that owns this IPC object.

**Owner read permission.** Whether the owner has read authority to the IPC object. Possible values follow:

0	The owner does not have read authority to the IPC object.
1	The owner has read authority to the IPC object.

**Owner write permission.** Whether the owner has write authority to the IPC object. Possible values follow:

0	The owner does not have write authority to the IPC object.
1	The owner has write authority to the IPC object.

**Reserved.** An ignored field.

**Resize.** Whether the shared memory object may be resized. Possible values follow:

0	The shared memory object may not be resized.
1	The shared memory object may be resized.

**Segment size.** The size of the shared memory segment.

**Size of all messages on queue.** The byte size of all of the messages that are currently on the queue.

**Teraspace.** Whether the shared memory object is attachable only to a process's teraspace. Possible values follow:

0	The shared memory object is not attachable to a process's teraspace.
1	The shared memory object is attachable to a process's teraspace.

**Title.** The title of the semaphore. The title contains the 16 characters that are associated with the semaphore when it is created.

**Value.** The value of the semaphore.

**Waiting qualified job identifier.** The job name, the job user profile, and the job number of a thread waiting on the semaphore. The 26 characters are:

1-10	The job name
11-20	The user profile
21-26	The job number

**Waiting thread identifier.** The thread ID of a thread waiting on the semaphore.

## Error Messages

Message ID	Error Message Text
CPF0F01 E	*SERVICE authority is required to run this program.
CPF2204 E	User profile &1 not found.
CPF24B4 E	Severe error while addressing parameter list.
CPF3C19 E	Error occurred with receiver variable specified.
CPF3C21 E	Format name &1 is not valid.
CPF3C90 E	Literal value cannot be changed.
CPF3CF1 E	Error code parameter not valid.
GUI0002 E	&2 is not valid for length of receiver variable.
GUI0027 E	&1 is not valid for number of records to return.
GUI0115 E	The list has been marked in error. See the previous messages.
GUI0118 E	Starting record cannot be 0 when records have been requested.
GUI0135 E	Filter key information is not valid.
GUI0136 E	Filter information is not valid.

API introduced: V4R2

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

---

## Open List of Semaphores (QP0ZOLSM) API

Required Parameter Group:

1	Receiver variable
<b>Output</b>	Char(*)
2	Length of receiver variable
<b>Input</b>	Binary(4)
3	List information
<b>Output</b>	Char(80)
4	Number of records to return
<b>Input</b>	Binary(4)
5	Format name
<b>Input</b>	Char(8)
6	Semaphore set identifier
<b>Input</b>	BINARY(4)
7	Error code
<b>I/O</b>	Char(*)
	Default Public Authority: *USE
Threadsafe: No	

The Open List of Semaphores (QP0ZOLSM) API lets you generate a list of description information about the semaphores within a semaphore set.

The QP0ZOLSM API places the specified number of list entries in the receiver variable. You can access additional records by using the Get List Entries (QGYGTLE) API. On successful completion of the QP0ZOLSM API, a handle is returned in the list information parameter. You may use this handle on subsequent calls to the following APIs:


- Get List Entries (QGYGTLE)
- Find Entry Number in List (QGYFNDE)
- Close List (QGYCLST)

The records returned by QP0ZOLSM include an information status field that describes the completeness and validity of the information. Be sure to check the information status field before using any other information returned.

## Authorities and Locks

### *Job Authority*

Service special authority (\*SERVICE) is needed to call this API.

For additional information on this authority, see the iSeries Security Reference  book.

## Required Parameter Group

### Receiver variable

OUTPUT; CHAR(\*)

The variable that is used to return the semaphore information that you requested.

### Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable.

### List information

OUTPUT; CHAR(80)

Information about the list of semaphores that were opened. For a description of the layout of this parameter, see Format of Open List Information.

### Number of records to return

INPUT; BINARY(4)

The number of records in the list to put into the receiver variable.

### Format name

INPUT; CHAR(8)

The format of the information to be returned in the receiver variable. You must use the following format name:

*LSEM0100* This format is described in "LSEM0100 Format."

### Semaphore set identifier

INPUT; BINARY(4)

The semaphore set identifier of the semaphore set whose semaphores you would like the information about. The semaphore set identifier can be obtained from calling either the `semget()`, or `QP0ZOLIP` API.

### Error code

I/O; CHAR(\*)

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

## LSEM0100 Format

This format name is used to return list information for the semaphores in a semaphore set. The following table shows the information returned in each record in the receiver variable for the LSEM0100 format. For a detailed description of each field, see "Field Descriptions" on page 46.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Length of entry
4	4	BINARY(4)	Number
8	8	BINARY(4)	Value
12	C	BINARY(4)	Displacement to wait values
16	10	BINARY(4)	Number of waiters
20	14	BINARY(4)	Size of waiting information
24	18	BINARY(4)	Waiting for zero

Offset		Type	Field
Dec	Hex		
28	1C	BINARY(4)	Waiting for positive value
32	20	CHAR(26)	Last changed qualified job identifier
58	3A	CHAR(2)	Reserved
60	3C	BINARY(4)	Process identifier
These fields repeat for each waiter on the semaphore value.		BINARY(4)	Wait value
		CHAR(26)	Waiting qualified job identifier
		CHAR(2)	Reserved

## Field Descriptions

**Displacement to wait values.** The offset in characters (bytes) from the beginning of the semaphore record to the beginning of the array of wait values.

**Last changed qualified job identifier.** The job name, the job user profile, and the job number of the thread that last changed the value of the semaphore. The 26 characters are:

1-10	The job name
11-20	The user profile
21-26	The job number

These fields will be all blanks if any of the following are true:

- No thread has changed the semaphore value.
- The process that changed the semaphore has ended.
- The process that changed the semaphore has not been initialized for signals.

**Length of entry.** The length of this semaphore record in the list.

**Number.** The semaphore number in the semaphore set.

**Number of waiters.** The total number of threads that are waiting for this semaphore to reach a certain value.

**Process identifier** The process identifier of the last thread to change the value of the semaphore. If no thread has changed the semaphore value, this field will be zero.

**Reserved.** An ignored field.

**Size of waiting information.** The size, in bytes, of the record that is used to store information about a thread that is waiting for a semaphore value.

**Value.** The current value of the semaphore.

**Wait value.** The value that a thread is waiting for the semaphore to reach. If the value is zero, the thread is waiting for the semaphore value to equal zero. If the value is a positive number, the thread is waiting for the semaphore value to be greater than or equal to this value.

**Waiting for positive value.** The number of threads that are currently waiting for a semaphore value to reach a positive number.

**Waiting for zero.** The number of threads that are currently waiting for the semaphore value to reach zero.

**Waiting qualified job identifier.** The job name, the job user profile, and the job number of the thread that is currently waiting for the semaphore. The 26 characters are:

1-10	The job name
11-20	The user profile
21-26	The job number

## Error Messages

Message ID	Error Message Text
GUI0002 E	&2 is not valid for length of receiver variable.
GUI0027 E	&1 is not valid for number of records to return.
GUI0115 E	The list has been marked in error. See the previous messages.
GUI0118 E	Starting record cannot be 0 when records have been requested.
CPF0F01 E	*SERVICE authority is required to run this program.
CPF2204 E	User profile &1 not found.
CPF24B4 E	Severe error while addressing parameter list.
CPF3C19 E	Error occurred with receiver variable specified.
CPF3C21 E	Format name &1 is not valid.
CPF3C90 E	Literal value cannot be changed.
CPF3CF1 E	Error code parameter not valid.
CPFA988 E	IPC object &1 does not exist.

API introduced: V4R2

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

---

## Retrieve an Interprocess Communication Object (QP0ZRIPC) API

Required Parameter Group:	
1	Receiver variable
<b>Output</b>	Char(*)
2	Length of receiver variable
<b>Input</b>	Binary(4)
3	Format name
<b>Input</b>	Char(8)
4	Identifier
<b>Input</b>	Binary(4)
5	Error code
<b>I/O</b>	Char(*)
Default Public Authority: *USE	
Threadsafe: No	


The Retrieve an Interprocess Communication Object (QP0ZRIPC) API lets you generate detailed information about a single interprocess communication (IPC) object. The object is identified by the format name and the identifier that is passed in.

The QP0ZRIPC API places the information about the object in the receiver variable. The information that is written to the receiver variable is dependent on the format name parameter.

## Authorities and Locks

### Job Authority

Service special authority (\*SERVICE) is needed to call this API.

For additional information on this authority, see the iSeries Security Reference  book.

## Required Parameter Group

### Receiver variable

OUTPUT; CHAR(\*)

The variable that is used to return the IPC object information that you requested.

### Length of receiver variable

INPUT; BINARY(4)

The length of the receiver variable. The minimum length is 8 bytes.

### Format name

INPUT; CHAR(8)

The format of the information to be returned in the receiver variable. This parameter will determine the object type (either message queues, semaphore sets, or shared memory) to retrieve the list for. You must use one of the following format names:

*RSST0100*      This format is described in "RSST0100 Format."  
*RMSQ0100*      This format is described in "RMSQ0100 Format" on page 49.  
*RSHM0100*      This format is described in "RSHM0100 Format" on page 50.

### Identifier

INPUT; BINARY(4)

The identifier of the IPC object that you would like to retrieve information about. This identifier is returned from the APIs `semget()`, `shmget()`, `msgget()`, or `QP0ZOLIP`.

### Error code

I/O; CHAR(\*)

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

## RSST0100 Format

This format name is used to return information for a single semaphore set. The following table shows the information returned in the receiver variable for the RSST0100 format. For a detailed description of each field, see "Field Descriptions" on page 51.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available



Offset		Type	Field
Dec	Hex		
8	8	BINARY(4)	Identifier
12	C	BINARY(4)	Key
16	10	BINARY(4)	Number of semaphores
20	14	CHAR(1)	Damaged
21	15	CHAR(1)	Owner read permission
22	16	CHAR(1)	Owner write permission
23	17	CHAR(1)	Group read permission
24	18	CHAR(1)	Group write permission
25	19	CHAR(1)	General read permission
26	1A	CHAR(1)	General write permission
27	1B	CHAR(1)	Authorized to delete
28	1C	CHAR(16)	Last semop() date and time
44	2C	CHAR(16)	Last administration change date and time
60	3C	CHAR(10)	Owner
70	46	CHAR(10)	Group owner
80	50	CHAR(10)	Creator
90	5A	CHAR(10)	Creator's group

## RMSQ0100 Format

This format name is used to return information about a single message queue. The following table shows the information returned in the receiver variable for the RMSQ0100 format. For a detailed description of each field, see "Field Descriptions" on page 51.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Identifier
12	C	BINARY(4)	Key
16	10	CHAR(1)	Damaged
17	11	CHAR(1)	Owner read permission
18	12	CHAR(1)	Owner write permission
19	13	CHAR(1)	Group read permission
20	14	CHAR(1)	Group write permission
21	15	CHAR(1)	General read permission
22	16	CHAR(1)	General write permission
23	17	CHAR(1)	Authorized to delete
24	18	BINARY(4)	Number of messages on queue
28	1C	BINARY(4)	Size of all messages on queue
32	20	BINARY(4)	Maximum size of all messages on queue

Offset		Type	Field
Dec	Hex		
36	24	BINARY(4)	Number of threads to receive message
40	28	BINARY(4)	Number of threads to send message
44	2C	CHAR(16)	Last msgrcv() date and time
60	3C	CHAR(16)	Last msgsnd() date and time
76	4C	CHAR(16)	Last administration change date and time
92	5C	CHAR(10)	Owner
102	66	CHAR(10)	Group owner
112	70	CHAR(10)	Creator
122	7A	CHAR(10)	Creator's group
132	84	CHAR(26)	Last msgsnd() qualified job identifier
158	9E	CHAR(2)	Reserved
160	A0	BINARY(4)	Last msgsnd() process identifier
164	A4	CHAR(26)	Last msgrcv() qualified job identifier
190	BE	CHAR(2)	Reserved
192	C0	BINARY(4)	Last msgrcv() process identifier
196	C4	BINARY(4)	Offset to message type
200	C8	BINARY(4)	Size of message information record
204	CC	BINARY(4)	Offset to wait type
208	D0	BINARY(4)	Size of message receive record
212	D4	BINARY(4)	Offset to wait size
216	D8	BINARY(4)	Size of message send record
These fields repeat for each message on queue.		BINARY(4)	Message type
		BINARY(4)	Message size
These fields repeat for each thread waiting to receive a message.		BINARY(4)	Message wait type
		CHAR(26)	Message receive qualified job identifier
		CHAR(2)	Reserved
These fields repeat for each thread waiting to send a message.		BINARY(4)	Message wait size
		CHAR(26)	Message send qualified job identifier
		CHAR(2)	Reserved

## RSHM0100 Format

This format name is used to return information for a single shared memory object. The following table shows the information returned in the receiver variable for the RSHM0100 format. For a detailed description of each field, see "Field Descriptions" on page 51.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available

Offset		Type	Field
Dec	Hex		
8	8	BINARY(4)	Identifier
12	C	BINARY(4)	Key
16	10	CHAR(1)	Damaged
17	11	CHAR(1)	Owner read permission
18	12	CHAR(1)	Owner write permission
19	13	CHAR(1)	Group read permission
20	14	CHAR(1)	Group write permission
21	15	CHAR(1)	General read permission
22	16	CHAR(1)	General write permission
23	17	CHAR(1)	Marked to be deleted
24	18	CHAR(1)	Authorized to delete
25	19	CHAR(1)	Teraspace
26	1A	CHAR(1)	Resize
27	1B	CHAR(1)	Reserved
28	1C	BINARY(4)	Segment size
32	20	BINARY(4)	Number attached
36	24	CHAR(16)	Last shmat() date and time
52	34	CHAR(16)	Last detach date and time
68	44	CHAR(16)	Last administration change date and time
84	54	CHAR(10)	Owner
94	5E	CHAR(10)	Group owner
104	68	CHAR(10)	Creator
114	72	CHAR(10)	Creator's group
124	7C	CHAR(26)	Last attach or detach qualified job identifier
150	96	CHAR(2)	Reserved
152	98	BINARY(4)	Last attach or detach process identifier
156	9C	BINARY(4)	Offset to times attached
160	A0	BINARY(4)	Number of attach entries
164	A4	BINARY(4)	Size of attach entry
These fields repeat for the number of attach entries.		BINARY(4)	Times attached
		CHAR(26)	Attached qualified job identifier
		CHAR(2)	Reserved

## Field Descriptions

**Attached qualified job identifier.** The job name, the job user profile, and the job number of a job that is attached to the shared memory segment. The 26 characters are:

- 1-10                   The job name
- 11-20                The user profile
- 21-26                The job number

**Authorized to delete.** This value determines if the caller has the authority to delete this IPC object. Possible values follow:

0                   The current thread cannot delete the IPC object.  
1                   The current thread can delete the IPC object.

**Bytes available.** The number of bytes of data available to be returned. All available data is returned if enough space is provided.

**Bytes returned.** The number of bytes of data returned.

**Creator.** The name of the user profile that created this IPC object.

**Creator's group.** The name of the group profile that created this IPC object. A special value can be returned:

\*NONE             The creator does not have a group profile.

**Damaged.** Whether the IPC object has suffered internal damage. Possible values follow:

0                   The IPC object is not damaged.  
1                   The IPC object is damaged.

**General read permission.** Whether any user other than the owner and group owner has read authority to the IPC object. Possible values follow:

0                   General read authority is not allowed to the IPC object.  
1                   General read authority is allowed to the IPC object.

**General write permission.** Whether any user other than the owner and group owner has write authority to the IPC object. Possible values follow:

0                   General write authority is not allowed to the IPC object.  
1                   General write authority is allowed to the IPC object.

**Group owner.** The name of the group profile that owns this IPC object. A special value can be returned:

\*NONE             The IPC object does not have a group owner.

**Group read permission.** Whether the group owner has read authority to the IPC object. Possible values follow:

0                   The group owner does not have read authority to the IPC object.  
1                   The group owner has read authority to the IPC object.

**Group write permission.** Whether the group owner has write authority to the IPC object. Possible values follow:

0                   The group owner does not have write authority to the IPC object.  
1                   The group owner has write authority to the IPC object.

**Identifier.** The unique IPC object identifier.

**Key.** The key of the IPC object. If this value is zero, this IPC object has no key associated with it.

**Last administration change date and time.** The date and time of the last change to the IPC object for the owner, group owner, or permissions. The 16 characters are:

1	Century, where 0 indicates years 19xx and 1 indicates years 20xx.
2-7	Date, in YYMMDD (year, month, and day) format.
8-13	Time of day, in HHMMSS (hours, minutes, and seconds) format.
14-16	Milliseconds.

**Last attach or detach process identifier.** The process identifier of the thread that performed the last successful attachment or detachment from the shared memory segment. If no thread has attached or detached from the shared memory segment, this field will be zero.

**Last attach or detach qualified job identifier.** The job name, the job user profile, and the job number of the thread that performed the last successful attachment or detachment from the shared memory segment. The 26 characters are:

1-10	The job name
11-20	The user profile
21-26	The job number

These fields will be all blanks if any of the following are true:

- No thread has performed an attachment or detachment on the shared memory.
- The last process that did an attachment or detachment on the shared memory has ended.
- The last process that did an attachment or detachment on the shared memory is not initialized for signals.

**Last detach date and time.** The date and time of the last detachment from the shared memory segment. If no thread has performed a successful detachment, this value will be set to all zeros. The 16 characters are:

1	Century, where 0 indicates years 19xx and 1 indicates years 20xx.
2-7	Date, in YYMMDD (year, month, and day) format.
8-13	Time of day, in HHMMSS (hours, minutes, and seconds) format.
14-16	Milliseconds.

**Last msgrcv() date and time.** The date and time of the last successful msgrcv() call. If no thread has performed a successful msgrcv() call, this value will be set to all zeros. The 16 characters are:

1	Century, where 0 indicates years 19xx and 1 indicates years 20xx.
2-7	Date, in YYMMDD (year, month, and day) format.
8-13	Time of day, in HHMMSS (hours, minutes, and seconds) format.
14-16	Milliseconds.

**Last msgrcv() process identifier.** The process identifier of the thread that performed the last successful msgrcv(). If no thread has done a msgrcv(), this field will be zero.

**Last msgrcv() qualified job identifier.** The job name, the job user profile, and the job number of the thread that performed the last successful msgrcv(). The 26 characters are:

1-10	The job name
11-20	The user profile
21-26	The job number

These fields will be all blanks if any of the following are true:

- No thread has received a message on this message queue.
- The last process to receive a message has ended.
- The last process to receive a message has not been initialized for signals.

**Last msgsnd() date and time.** The date and time of the last successful msgsnd() call. If no thread has performed a successful msgsnd() call, this value will be set to all zeros. The 16 characters are:

1	Century, where 0 indicates years 19xx and 1 indicates years 20xx.
2-7	Date, in YYMMDD (year, month, and day) format.
8-13	Time of day, in HHMMSS (hours, minutes, and seconds) format.
14-16	Milliseconds.

**Last msgsnd() process identifier.** The process identifier of the thread that performed the last successful msgsnd(). If no thread has done a msgsnd(), this field will be zero.

**Last msgsnd() qualified job identifier.** The job name, the job user profile, and the job number of the thread that performed the last successful msgsnd(). The 26 characters are:

1-10	The job name
11-20	The user profile
21-26	The job number

These fields will be all blanks if any of the following are true:

- No thread has sent a message to this message queue.
- The last process to send a message has ended.
- The last process to send a message has not been initialized for signals.

**Last semop() date and time.** The date and time of the last successful semop() call. If no thread has performed a successful semop() call, this value will be set to all zeros. The 16 characters are:

1	Century, where 0 indicates years 19xx and 1 indicates years 20xx.
2-7	Date, in YYMMDD (year, month, and day) format.
8-13	Time of day, in HHMMSS (hours, minutes, and seconds) format.
14-16	Milliseconds.

**Last shmat() date and time.** The date and time of the last successful shmat(). If no thread has performed a successful shmat() call, this value will be set to all zeros. The 16 characters are:

1	Century, where 0 indicates years 19xx and 1 indicates years 20xx.
2-7	Date, in YYMMDD (year, month, and day) format.
8-13	Time of day, in HHMMSS (hours, minutes, and seconds) format.
14-16	Milliseconds.

**Marked to be deleted.** Whether the shared memory is marked to be deleted when the number attached becomes zero. Possible values follow:

- 0 The shared memory segment is not marked for deletion.
- 1 The shared memory segment is marked for deletion.

**Maximum size of all messages on queue.** The maximum byte size of all messages that can be on the queue at one time.

**Message receive qualified job identifier.** The job name, the job user profile, and the job number of the thread that is waiting to receive a message. The 26 characters are:

- 1-10 The job name
- 11-20 The user profile
- 21-26 The job number

**Message send qualified job identifier.** The job name, the job user profile, and the job number of the thread that is waiting to send a message. The 26 characters are:

- 1-10 The job name
- 11-20 The user profile
- 21-26 The job number

**Message size.** The message size of a message that is currently on the queue.

**Message type.** The message type of a message that is currently on the queue.

**Message wait size.** The message size of a message that a thread is currently waiting to put on the queue.

**Message wait type.** The message type that a thread is currently waiting to receive.

**Number attached.** The number of times any thread has done a shmat() without doing a detach. One process can be attached multiple times to the same shared memory segment.

**Number of attach entries.** The number of entries in the variable length section of RSHM0100.

**Number of threads to receive message.** The number of threads that are currently waiting to receive a message.

**Number of threads to send message.** The number of threads that are currently waiting to send a message.

**Number of messages on queue.** The number of messages that are currently on the message queue.

**Number of semaphores.** The number of semaphores in the semaphore set.

**Offset to message type.** The offset in characters (bytes) from the beginning of the RMSQ0100 record to the message type field.

**Offset to times attached.** The offset in characters (bytes) from the beginning of the RSHM0100 record to the times attached field.

**Offset to wait size.** The offset in characters (bytes) from the beginning of the RMSQ0100 record to the wait size field.

**Offset to wait type.** The offset in characters (bytes) from the beginning of the RMSQ0100 record to the wait type field.

**Owner.** The name of the user profile that owns this IPC object.

**Owner read permission.** Whether the owner has read authority to the IPC object. Possible values follow:

0 The owner does not have read authority to the IPC object.  
1 The owner has read authority to the IPC object.

**Owner write permission.** Whether the owner has write authority to the IPC object. Possible values follow:

0 The owner does not have write authority to the IPC object.  
1 The owner has write authority to the IPC object.

**Reserved.** An ignored field.

**Resize.** Whether the shared memory object may be resized. Possible values follow:

0 The shared memory object may not be resized.  
1 The shared memory object may be resized.

**Segment size.** The size of the shared memory segment.

**Size of all messages on queue.** The size, in bytes, of all of the messages that are currently on the queue.

**Size of attach entry.** The size, in bytes, of each attach entry in the array of attach entries.

**Size of message information record.** The size, in bytes, of each message information record.

**Size of message receive record.** The size, in bytes, of the record that is used to store information about a thread waiting to receive a message.

**Size of message send record.** The size, in bytes, of the record that is used to store information about a thread waiting to send a message.

**Teraspace.** Whether the shared memory object is attachable only to a process's teraspace. Possible values follow:

0 The shared memory object is not attachable to a process's teraspace.  
1 The shared memory object is attachable to a process's teraspace.

**Times attached.** The number of times that this process is attached to the shared memory.

## Error Messages

Message ID	Error Message Text
GUI0002 E	&2 is not valid for length of receiver variable.
CPF0F01 E	*SERVICE authority is required to run this program.
CPF24B4 E	Severe error while addressing parameter list.
CPF3C19 E	Error occurred with receiver variable specified.
CPF3C21 E	Format name &1 is not valid.
CPF3C90 E	Literal value cannot be changed.



Message ID	Error Message Text
CPF3CF1 E	Error code parameter not valid.
CPFA988 E	IPC object &1 does not exist.

API introduced: V4R2

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

---

## semctl()—Perform Semaphore Control Operations

```
Syntax
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);

Service Program Name: QP0ZCPA

Default Public Authority: *USE

Threadsafe: Yes
```

The **semctl()** function allows the caller to control the semaphore set specified by the *semid* parameter.

A semaphore set is controlled by setting the *cmd* parameter to one of the following values:

### IPC\_RMID (0x00000000)

Remove the semaphore set identifier *semid* from the system and destroy the set of semaphores. Any threads that are waiting in **semop()** are woken up and **semop()** returns with a return value of -1 and *errno* set to EIDRM.

The IPC\_RMID command can be run only by a thread with appropriate privileges or one that has an effective user ID equal to the user ID of the owner or the user ID of the creator of the semaphore set.

### IPC\_SET (0x00000001)

Set the user ID of the owner, the group ID of the owner, and the permissions for the semaphore set to the values in the *sem\_perm.uid*, *sem\_perm.gid*, and *sem\_perm.mode* members of the *semid\_ds* data structure pointed to by the fourth parameter.

The IPC\_SET command can be run only by a thread with appropriate privileges or one that has an effective user ID equal to the user ID of the owner or the user ID of the creator of the semaphore set.

### IPC\_STAT (0x00000002)

Store the current value of each member of the *semid\_ds* data structure into the structure pointed to by the fourth parameter. The IPC\_STAT command requires read permission to the semaphore set.

### GETNCNT (0x00000003)

Return the number of threads waiting for the value of semaphore *semnum* to increase. This value is the *semncnt* value in the *semaphore\_t* data structure associated with the specified semaphore. The GETNCNT command requires read permission to the semaphore set.

### GETPID (0x00000004)

Return the process ID of the last thread to operate on semaphore *semnum*. This value is the *sempid* value in the *semaphore\_t* data structure associated with the specified semaphore. The GETPID command requires read permission to the semaphore set.

**GETVAL (0x00000005)**

Return the current value of semaphore *semnum*. This value is the *semval* value in the *semaphore\_t* data structure associated with the specified semaphore. The GETVAL command requires read permission to the semaphore set.

**GETALL (0x00000006)**

Return the values of each semaphore in the semaphore set into the array pointed to by the fourth parameter, which is a pointer to an array of type *unsigned short*. The values are the *semval* value in the *semaphore\_t* data structure associated with each semaphore in the semaphore set. The GETALL command requires read permission to the semaphore set.

**GETZCNT (0x00000007)**

Return the number of threads waiting for the value of semaphore *semnum* to reach zero. This value is the *semzcnt* value in the *semaphore\_t* data structure associated with the specified semaphore. The GETZCNT command requires read permission to the semaphore set.

**SETVAL (0x00000008)**

Set the value of semaphore *semnum* to the integer value of type *int* specified in the fourth parameter and clear the associated per-thread semaphore adjustment value. The SETVAL command requires write permission to the semaphore set.

**SETALL (0x00000009)**

Set the values of each semaphore in the semaphore set to the values contained in the array pointed to by the fourth parameter, which is a pointer to an array of type *unsigned short*. In addition, the associated per-thread semaphore-adjustment value is cleared for each semaphore. The SETALL command requires write permission to the semaphore set.

## Parameters

**semid** (Input) Semaphore set identifier, a positive integer. It is returned by the “*semget()*—Get Semaphore Set with Key” on page 62 function and used to identify the semaphore set on which to perform the control operation.

**semnum**

(Input) Semaphore number, a non-negative integer. It identifies a semaphore within the semaphore set on which to perform the control operation.

**cmd** (Input) Command, the control operation to perform on the semaphore set. Valid values are listed above.

... (Input/output) An optional fourth parameter whose type depends on the value of *cmd*. For the *cmd* SETVAL, this parameter must be an integer of type *int*. For the *cmd* IPC\_STAT or IPC\_SET, this parameter must be a pointer to a *semid\_ds* structure. For the *cmd* GETALL or SETALL, this parameter must be a pointer to an array of type *unsigned short*. For all other values of *cmd*, this parameter is not required.

The members of the *semid\_ds* structure are as follows:

*struct ipc\_perm* The members of the `ipc_perm` structure are as follows:

*sem\_perm*

*uid\_t uid*  
The user ID of the owner of the semaphore set.

*gid\_t gid*  
The group ID of the owner of the semaphore set.

*uid\_t cuid*  
The user ID of the creator of the semaphore set.

*gid\_t cgid*  
The group ID of the creator of the semaphore set.

*mode\_t mode*  
The permissions for the semaphore set.

*unsigned short sem\_nsems*  
The number of semaphores in the set.

*time\_t sem\_otime* The time the last thread operated on the semaphore set using `semop()`.

*time\_t sem\_ctime* The time the last thread changed the semaphore set using `semctl()`.

## Authorities

### Authorization Required for `semctl()`

Object Referred to	Authority Required	errno
Semaphore, get the value of ( <i>cmd</i> = GETVAL)	Read	EACCES
Semaphore, set the value of ( <i>cmd</i> = SETVAL)	Write	EACCES
Semaphore, get last process to operate on ( <i>cmd</i> = GETPID)	Read	EACCES
Semaphore, get number of threads waiting for value to increase ( <i>cmd</i> = GETNCNT)	Read	EACCES
Semaphore, get number of threads waiting for value to reach zero ( <i>cmd</i> = GETZCNT)	Read	EACCES
Semaphore set, get value of each semaphore ( <i>cmd</i> = GETALL)	Read	EACCES
Semaphore set, set value of each semaphore ( <i>cmd</i> = SETALL)	Write	EACCES
Semaphore set, retrieve state information ( <i>cmd</i> = IPC_STAT)	Read	EACCES
Semaphore set, set state information ( <i>cmd</i> = IPC_SET)	See Note	EPERM
Semaphore set, remove ( <i>cmd</i> = IPC_RMID)	See Note	EPERM

**Note:** To set semaphore set information or to remove a semaphore set, the thread must be the owner or creator of the semaphore set, or have appropriate privileges.

## Return Value

<i>value</i>	<b>semctl()</b> was successful. Depending on the control operation specified in <i>cmd</i> , <b>semctl()</b> returns the following values:  <i>GETVAL</i> The value of the specified semaphore.  <i>GETPID</i> The process ID of the last thread that performed a semaphore operation on the specified semaphore.  <i>GETNCNT</i> The number of threads waiting for the value of the specified semaphore to increase.  <i>GETZCNT</i> The number of threads waiting for the value of the specified semaphore to reach zero.  <i>For all other values of cmd:</i> The value is 0.
-1	<b>semctl()</b> was not successful. The <i>errno</i> variable is set to indicate the error.

## Error Conditions

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

### [EACCES]

Permission denied.

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

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

The *cmd* parameter is *IPC\_STAT*, *GETVAL*, *GETPID*, *GETNCNT*, *GETZCNT*, or *GETALL* and the calling thread does not have read permission to the semaphore set.

The *cmd* parameter is *SETVAL*, or *SETALL* and the calling thread does not have write permission to the semaphore set.

### [EDAMAGE]

A damaged object was encountered.

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

The semaphore set has been damaged by a previous semaphore operation.

### [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.

One of the following has occurred:

- The *semid* parameter is not a valid semaphore identifier.
- The *semnum* parameter is less than zero or greater than or equal to `sem_nsems`.
- The *cmd* parameter is not a valid command.

#### [EPERM]

Operation not permitted.

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

The *cmd* parameter is `IPC_RMID` or `IPC_SET` and both of the following are true:

- the calling thread does not have appropriate privileges.
- the effective user ID of the calling thread is not equal to the user ID of the owner or the user ID of the creator of the semaphore set.

#### [ERANGE]

A range error occurred.

The value of an argument is too small, or a result too large.

The *cmd* parameter is `SETVAL`, and the value to which `semval` is to be set is greater than the system-imposed maximum.

#### [EUNKNOWN]

Unknown system state.

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

## Error Messages

None.

## Usage Notes

1. “Appropriate privileges” is defined to be `*ALLOBJ` special authority. If the user profile under which the thread is running does not have `*ALLOBJ` special authority, the thread does not have appropriate privileges.
2. Take care when using a union for the optional fourth parameter. If the optional fourth parameter is an integer, `semctl()` expects it to directly follow the third parameter in storage. But a union that contains a pointer is aligned on a 16-byte boundary, which might not directly follow the third parameter. Therefore, the value used by `semctl()` for the fourth parameter might not be the value intended by the caller, and unexpected results could occur.

## Related Information

- The `<sys/sem.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`semget()`—Get Semaphore Set with Key” on page 62—Get Semaphore Set with Key
- “`semop()`—Perform Semaphore Operations on Semaphore Set” on page 65—Perform Semaphore Operations on Semaphore Set

## Example

See Code disclaimer information for information pertaining to code examples.

For an example of using this function, see Using Semaphores and Shared Memory in Examples: APIs.

---

## semget()—Get Semaphore Set with Key

**Syntax**

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

int semget(key_t key, int nsems, int semflg);
```

Service Program Name: QP0ZCPA

Default Public Authority: \*USE

Threadsafe: Yes

The **semget()** function either creates a new semaphore set or returns the semaphore set identifier associated with the *key* parameter for an existing semaphore set. A new semaphore set is created if one of the following is true:

- The *key* parameter is equal to `IPC_PRIVATE`.
- The *key* parameter does not already have a semaphore set identifier associated with it and the `IPC_CREAT` flag is specified in the *semflg* parameter.

The system maintains status information about a semaphore set which can be retrieved with the “**semctl()**—Perform Semaphore Control Operations” on page 57 function. When a new semaphore set is created, the system initializes the members of the `semid_ds` structure as follows:

- `sem_perm.cuid` and `sem_perm.uid` are set to the current user ID of the thread.
- `sem_perm.cgid` and `sem_perm.gid` are set to the current group ID of the thread.
- The low-order 9 bits of `shm_perm.mode` are set equal to the low-order 9 bits of the *shmflg* parameter.
- `sem_nsems` is set to the value specified in the *nsems* parameter.
- `sem_ctime` is set to the current time.
- `sem_otime` is set to zero.

**Parameters**

**key** (Input) Key associated with the semaphore set. A key of `IPC_PRIVATE` (0x00000000) guarantees that a unique semaphore set is created. A key can also be specified by the caller or generated by the “**ftok()**—Generate IPC Key from File Name” on page 3 function.

**nsems** (Input) Number of semaphores in the semaphore set. The number of semaphores in the set cannot be changed after the semaphore set is created. If an existing semaphore set is being accessed, *nsems* can be zero.

**semflg**

(Input) Operations and permission flags. The *semflg* parameter value is either zero, or is obtained by performing an OR operation on one or more of the following constants:

**S\_IRUSR (0x00000100)**

Allow the owner of the semaphore set to read from it.

**S\_IWUSR (0x00000080)**

Allow the owner of the semaphore set to write to it.

**S\_IRGRP (0x00000020)**

Allow the group of the semaphore set to read from it.

**S\_IWGRP (0x00000010)**

Allow the group of the semaphore set to write to it.

**S\_IROTH (0x00000004)**

Allow others to read from the semaphore set.

**S\_IWOTH (0x00000002)**

Allow others to write to the semaphore set.

**IPC\_CREAT (0x00000200)**

Create the semaphore set if it does not exist.

**IPC\_EXCL (0x00000400)**

Return an error if the IPC\_CREAT flag is set and the semaphore set already exists.

## Authorities

### Authorization Required for semget()

Object Referred to	Authority Required	errno
Semaphore set to be created	None	None
Existing semaphore set to be accessed	See Note	EACCES

**Note:** If the thread is accessing a semaphore set that already exists, the mode specified in the last 9 bits of *semflg* must be a subset of the mode of the existing semaphore set.

## Return Value

*value*                **semget()** was successful. The value returned is the semaphore set identifier associated with the *key* parameter.

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

## Error Conditions

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

### [EACCES]

Permission denied.

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

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

A semaphore set identifier exists for the parameter *key*, but permissions specified in the low-order 9 bits of *semflg* are not a subset of the current permissions.

### [EDAMAGE]

A damaged object was encountered.

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

The semaphore set has been damaged by a previous semaphore operation.

#### [EEXIST]

File exists.

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

The named file, directory, or path already exists.

A semaphore identifier exists for the *key* parameter, and both the `IPC_CREAT` and `IPC_EXCL` flags are set in the *semflg* parameter.

#### [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.

One of the following has occurred:

- The value of *nsems* is either less than or equal to zero, or greater than the system-imposed limit.
- A semaphore identifier exists for the parameter *key*, but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not zero.

#### [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.

A semaphore set identifier does not exist for the *key* parameter, and the `IPC_CREAT` flag is not set in the *semflg* parameter.

#### [ENOSPC]

No space available.

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

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

A semaphore set identifier cannot be created because the system limit on the maximum number of allowed semaphore set identifiers would be exceeded.

#### [EUNKNOWN]

Unknown system state.

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

## Error Messages

None.

## Usage Notes

1. The best way to generate a unique key is to use the “`ftok()`—Generate IPC Key from File Name” on page 3 function.



2. A “semctl()—Perform Semaphore Control Operations” on page 57 call specifying a *cmd* parameter of SETALL should be used to initialize the semaphore values after the semaphore set is created.

## Related Information

- The <sys/sem.h> file (see “Header Files for UNIX-Type Functions” on page 144)
- “ftok()—Generate IPC Key from File Name” on page 3—Generate IPC Key from File Name
- “semctl()—Perform Semaphore Control Operations” on page 57—Perform Semaphore Control Operations
- “semop()—Perform Semaphore Operations on Semaphore Set”—Perform Semaphore Operations on Semaphore Set

## Example

See Code disclaimer information for information pertaining to code examples.

For an example of using this function, see Using Semaphores and Shared Memory in Examples: APIs.

API introduced: V3R6

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

---

## semop()—Perform Semaphore Operations on Semaphore Set

Syntax

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

```
int semop(int semid, struct sembuf *sops,  
          size_t nsops);
```

Service Program Name: QP0ZCPA

Default Public Authority: \*USE

Threadsafe: Yes

The **semop()** function performs operations on semaphores in a semaphore set. These operations are supplied in a user-defined array of operations.

Each semaphore operation specified by the *sops* array is performed on the semaphore set specified by *semid*. The entire array of operations is performed atomically; no other thread will operate on the semaphore set until all of the operations are done or it is determined that they cannot be done. If the entire set of operations cannot be performed, none of the operations are done, and the thread waits until all of the operations can be done.

The members of the *sembuf* structure are as follows:

<i>unsigned short sem_num</i>	The number of the semaphore in the semaphore set.
<i>short sem_op</i>	The operation to perform on the semaphore.
<i>short sem_flg</i>	The operation flags.

The **semop()** function changes each semaphore specified by *sem\_num* according to the value of *sem\_op* as follows:

- If *sem\_op* is positive, **semop()** increments the value of the semaphore and wakes up any threads waiting for the semaphore to increase. This corresponds to releasing resources controlled by the semaphore.
- If *sem\_op* is negative, **semop()** attempts to decrement the value of the semaphore. If the result would be negative, it waits for the semaphore value to increase. If the result would be positive, it decrements the semaphore. If the result would be zero, it decrements the semaphore and wakes up any threads waiting for the semaphore to be zero. This corresponds to the allocation of resources.
- If *sem\_op* is zero, the thread waits for the semaphore's value to be zero.

If `IPC_NOWAIT` is set in *sem\_flg* and the operation cannot be completed, **semop()** returns with a return value of -1 and *errno* set to `EAGAIN` instead of causing the thread to wait.

If `SEM_UNDO` is set in *sem\_flg*, **semop()** causes IPC to reverse the effect of this semaphore operation when the thread ends, effectively releasing the resources or request for resources controlled by the semaphore. This value is known as the semaphore adjustment value.

If the thread waits for the semaphore value to change, the calling thread suspends processing until one of the following occurs:

- The semaphore reaches the specified value.
- The semaphore set identifier *semid* is removed from the system. When this occurs, the **semop()** function returns with a return value of -1 and *errno* set to `EIDRM`.
- A signal is delivered to the calling thread. When this occurs, the **semop** function returns with a return value of -1 and *errno* set to `EINTR`.

The system maintains status information about a semaphore set which can be retrieved with the “`semctl()`—Perform Semaphore Control Operations” on page 57 function. When a semaphore operation is successfully completed, the system sets the members of the `semid_ds` structure as follows:

- `shmotime` is set to the current time.

## Parameters

*semid* (Input) Semaphore set identifier, a positive integer. It is returned by the “`semget()`—Get Semaphore Set with Key” on page 62 function and used to identify the semaphore set on which to perform the control operation.

*sops* (Input) Pointer to array of semaphore operation structures.

*nsops* (Input) Number of `sembuf` structures in *sops* array.

## Authorities

### Authorization Required for `semop()`

Object Referred to	Authority Required	errno
Semaphore, <i>sem_op</i> is negative	Write	EACCES
Semaphore, <i>sem_op</i> is positive	Write	EACCES
Semaphore, <i>sem_op</i> is zero	Read	EACCES

## Return Value

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

## Error Conditions

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

### [EACCES]

Permission denied.

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

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

The *sem\_op* value is negative or positive and the calling thread does not have write permission to the semaphore set.

The *sem\_op* value is zero and the calling thread does not have read permission to the semaphore set.

### [EAGAIN]

Operation would have caused the process to be suspended.

The operation would result in the calling thread waiting and the `IPC_NOWAIT` flag is set in the *sem\_flg* member.

### [EDAMAGE]

A damaged object was encountered.

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

The semaphore set has been damaged by a previous semaphore operation.

### [EFAULT]

The address used for an argument is not correct.

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

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

### [EFBIG]

Object is too large.

The size of the object would exceed the system allowed maximum size.

The *sem\_num* parameter is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*.

### [EIDRM]

ID has been removed.

The semaphore identifier *semid* has been removed from the system.

### [EINTR]

Interrupted function call.

The *semop()* function was interrupted by a signal.

### [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 *semid* parameter is not a valid semaphore identifier.

### [ENOSPC]

No space available.

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

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

The limit on the number of individual threads requesting a SEM\_UNDO would be exceeded.

### [ERANGE]

A range error occurred.

The value of an argument is too small, or a result too large.

An operation would cause a *semval* to overflow the system-imposed limit, or an operation would cause a semaphore adjustment value to overflow the system-imposed limit.

### [EUNKNOWN]

Unknown system state.

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

## Error Messages

None.

## Usage Notes

### Related Information

- The `<sys/sem.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`semget()`—Get Semaphore Set with Key” on page 62—Get Semaphore Set with Key
- “`semctl()`—Perform Semaphore Control Operations” on page 57—Perform Semaphore Control Operations

### Example

See Code disclaimer information for information pertaining to code examples.

For an example of using this function, see Using Semaphores and Shared Memory in Examples: APIs.

API introduced: V3R6

Top | UNIX-Type APIs | APIs by category

---

## sem\_close()—Close Named Semaphore

### Syntax

```
#include <semaphore.h>
```

```
int sem_close(sem_t * sem);
```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes

The **sem\_close()** function closes a named semaphore that was previously opened by a thread of the current process using **sem\_open()** or **sem\_open\_np()**. The **sem\_close()** function frees system resources associated with the semaphore on behalf of the process. Using a semaphore after it has been closed will result in an error. A semaphore should be closed when it is no longer used. If a **sem\_unlink()** was performed previously for the semaphore and the current process holds the last reference to the semaphore, then the named semaphore will be deleted and removed from the system.

## Parameters

**sem** (Input) A pointer to an opened named semaphore. This semaphore is closed for this process.

## Authorities

No authorization is required. Authorization is verified during **sem\_open()**.

## Return Value

0 **sem\_close()** was successful.

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

## Error Conditions

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

### [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 *sem* parameter is not a valid semaphore.

## Error Messages

None.

## Related Information

- The `<semaphore.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`sem_getvalue()`—Get Semaphore Value” on page 72—Get Semaphore Value
- “`sem_open()`—Open Named Semaphore” on page 78—Open Named Semaphore
- “`sem_open_np()`—Open Named Semaphore with Maximum Value” on page 81—Open Named Semaphore with Maximum Value
- “`sem_post()`—Post to Semaphore” on page 85—Post to Semaphore
- “`sem_post_np()`—Post Value to Semaphore” on page 86—Post Value to Semaphore
- “`sem_trywait()`—Try to Decrement Semaphore” on page 89—Try to Decrement Semaphore
- “`sem_unlink()`—Unlink Named Semaphore” on page 91—Unlink Named Semaphore
- “`sem_wait()`—Wait for Semaphore” on page 93—Wait for Semaphore
- “`sem_wait_np()`—Wait for Semaphore with Timeout” on page 95—Wait for Semaphore with Timeout

## Example

See Code disclaimer information for information pertaining to code examples.

The following example opens a named semaphore with an initial value of 10 and then closes it.

```
#include <semaphore.h>
main() {
    sem_t * my_semaphore;
    int rc;

    my_semaphore = sem_open("/mysemaphore",
                          O_CREAT, S_IRUSR | S_IWUSR,
                          10);

    sem_close(my_semaphore);
}
```

API introduced: V4R4

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

---

## `sem_destroy()`—Destroy Unnamed Semaphore

Syntax

```
#include <semaphore.h>

int sem_destroy(sem_t * sem);
```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes

The `sem_destroy()` function destroys an unnamed semaphore that was previously initialized using `sem_init()` or `sem_init_np()`. Any threads that have blocked from calling `sem_wait()` or `sem_wait_np()` on the semaphore will unblock and return an [EINVAL] or [EDESTROYED] error.

## Parameters

*sem* (Input) A pointer to an initialized unnamed semaphore. The semaphore is destroyed.

## Authorities

None

## Return Value

0 **sem\_destroy()** was successful.  
-1 **sem\_destroy()** was not successful. The *errno* variable is set to indicate the error.

## Error Conditions

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

### [EBUSY]

Resource busy.

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

The semaphore is being destroyed by another thread.

### [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 *sem* parameter is not a valid semaphore.

## Error Messages

None.

## Related Information

- The <**semaphore.h**> file (see “Header Files for UNIX-Type Functions” on page 144)
- “**sem\_getvalue()**—Get Semaphore Value” on page 72—Get Semaphore Value
- “**sem\_init()**—Initialize Unnamed Semaphore” on page 74—Initialize Unnamed Semaphore
- “**sem\_init\_np()**—Initialize Unnamed Semaphore with Maximum Value” on page 75—Initialize Unnamed Semaphore with Maximum Value
- “**sem\_post()**—Post to Semaphore” on page 85—Post to Semaphore
- “**sem\_post\_np()**—Post Value to Semaphore” on page 86—Post Value to Semaphore
- “**sem\_trywait()**—Try to Decrement Semaphore” on page 89—Try to Decrement Semaphore
- “**sem\_unlink()**—Unlink Named Semaphore” on page 91—Unlink Named Semaphore
- “**sem\_wait()**—Wait for Semaphore” on page 93—Wait for Semaphore
- “**sem\_wait\_np()**—Wait for Semaphore with Timeout” on page 95—Wait for Semaphore with Timeout

## Example

See Code disclaimer information for information pertaining to code examples.

The following example initializes an unnamed semaphore, `my_semaphore`, that will be used by threads of the current process and sets its value to 10. The semaphore is then destroyed using `sem_destroy()`.

```
#include <semaphore.h>
main() {
    sem_t my_semaphore;
    int rc;

    rc = sem_init(&my_semaphore, 0, 10);
    rc = sem_destroy(&my_semaphore);
}
```

API introduced: V4R4

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

---

## sem\_getvalue()—Get Semaphore Value

```
Syntax
#include <semaphore.h>

int sem_getvalue(sem_t * sem, int * value);

Service Program Name: QP0ZPSEM

Default Public Authority: *USE

Threadsafe: Yes
```

The `sem_getvalue()` function retrieves the value of a named or unnamed semaphore. If the current value of the semaphore is zero and there are threads waiting on the semaphore, a negative value is returned. The absolute value of this negative value is the number of threads waiting on the semaphore.

### Parameters

**sem** (Input) A pointer to an initialized unnamed semaphore or an opened named semaphore.  
**value** (Output) A pointer to the integer that contains the value of the semaphore.

### Authorities

None

### Return Value

0 `sem_getvalue()` was successful.  
-1 `sem_getvalue()` was not successful. The `errno` variable is set to indicate the error.

### Error Conditions

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

[EINVAL]  
The 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.

## Error Messages

None.

## Related Information

- The `<semaphore.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`sem_close()`—Close Named Semaphore” on page 69—Close Named Semaphore
- “`sem_destroy()`—Destroy Unnamed Semaphore” on page 70—Destroy Unnamed Semaphore
- “`sem_init()`—Initialize Unnamed Semaphore” on page 74—Initialize Unnamed Semaphore
- “`sem_init_np()`—Initialize Unnamed Semaphore with Maximum Value” on page 75—Initialize Unnamed Semaphore with Maximum Value
- “`sem_open()`—Open Named Semaphore” on page 78—Open Named Semaphore
- “`sem_open_np()`—Open Named Semaphore with Maximum Value” on page 81—Open Named Semaphore with Maximum Value
- “`sem_post()`—Post to Semaphore” on page 85—Post to Semaphore
- “`sem_post_np()`—Post Value to Semaphore” on page 86—Post Value to Semaphore
- “`sem_trywait()`—Try to Decrement Semaphore” on page 89—Try to Decrement Semaphore
- “`sem_unlink()`—Unlink Named Semaphore” on page 91—Unlink Named Semaphore
- “`sem_wait()`—Wait for Semaphore” on page 93—Wait for Semaphore
- “`sem_wait_np()`—Wait for Semaphore with Timeout” on page 95—Wait for Semaphore with Timeout

## Example

See Code disclaimer information for information pertaining to code examples.

The following example retrieves the value of a semaphore before and after it is decremented by `sem_wait()`.

```
#include <stdio.h>
#include <semaphore.h>
main() {
    sem_t my_semaphore;
    int value;

    sem_init(&my_semaphore, 0, 10);
    sem_getvalue(&my_semaphore, &value);
    printf("The initial value of the semaphore is %d\n", value);
    sem_wait(&my_semaphore);
    sem_getvalue(&my_semaphore, &value);
    printf("The value of the semaphore after the wait is %d\n", value);
}
```

## Output:

```
The initial value of the semaphore is 10
The value of the semaphore after the wait is 9
```

API introduced: V4R4

---

## sem\_init()—Initialize Unnamed Semaphore

### Syntax

```
#include <semaphore.h>

int sem_init(sem_t * sem, int shared,
             unsigned int value);
```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes

The **sem\_init()** function initializes an unnamed semaphore and sets its initial value. The maximum value of the semaphore is set to SEM\_VALUE\_MAX. The title for the semaphore is set to the character representation of the address of the semaphore. If an unnamed semaphore already exists at *sem*, then it will be destroyed and a new semaphore will be initialized.

## Parameters

**sem** (Input) A pointer to the storage of an uninitialized unnamed semaphore. The pointer must be aligned on a 16-byte boundary. This semaphore is initialized.

### shared

(Input) An indication to the system of how the semaphore is going to be used. A value of zero indicates that the semaphore will be used only by threads within the current process. A nonzero value indicates that the semaphore may be used by threads from other processes.

**value** (Input) The value used to initialize the value of the semaphore.

## Authorities

None

## Return Value

0                    **sem\_init()** was successful.  
-1                   **sem\_init()** was not successful. The *errno* variable is set to indicate the error.

## Error Conditions

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

### [EINVAL]

The value specified for the argument is not correct.

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

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

The *value* parameter is greater than SEM\_VALUE\_MAX.

### [ENOSPC]

No space available.

System semaphore resources have been exhausted.

## Error Messages

None.

## Related Information

- The `<semaphore.h>` file (see )
- “`sem_destroy()`—Destroy Unnamed Semaphore” on page 70—Destroy Unnamed Semaphore
- “`sem_getvalue()`—Get Semaphore Value” on page 72—Get Semaphore Value
- “`sem_init_np()`—Initialize Unnamed Semaphore with Maximum Value”—Initialize Unnamed Semaphore with Maximum Value
- “`sem_post()`—Post to Semaphore” on page 85—Post to Semaphore
- “`sem_post_np()`—Post Value to Semaphore” on page 86—Post Value to Semaphore
- “`sem_trywait()`—Try to Decrement Semaphore” on page 89—Try to Decrement Semaphore
- “`sem_wait()`—Wait for Semaphore” on page 93—Wait for Semaphore
- “`sem_wait_np()`—Wait for Semaphore with Timeout” on page 95—Wait for Semaphore with Timeout

## Example

See Code disclaimer information for information pertaining to code examples.

The following example initializes an unnamed semaphore, `my_semaphore`, that will be used by threads of the current process. Its value is set to 10.

```
#include <semaphore.h>
main() {
    sem_t my_semaphore;
    int rc;

    rc = sem_init(&my_semaphore, 0, 10);
}
```

API introduced: V4R4

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

---

## `sem_init_np()`—Initialize Unnamed Semaphore with Maximum Value

Syntax

```
#include <semaphore.h>

int sem_init_np(sem_t * sem, int shared,
               unsigned int value,
               sem_attr_np_t * attr);
```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes

The `sem_init_np()` function initializes an unnamed semaphore and sets its initial value. The `sem_init_np()` function uses the `attr` parameter to set the maximum value and title of the semaphore. If an unnamed semaphore already exists at `sem`, then it will be destroyed and a new semaphore will be initialized.

## Parameters

**sem** (Input) A pointer to the storage of an uninitialized unnamed semaphore. The pointer must be aligned on a 16-byte boundary. This semaphore is initialized.

### shared

(Input) An indication to the system of how the semaphore is going to be used. A value of zero indicates that the semaphore will be used only by threads within the current process. A nonzero value indicates that the semaphore may be used by threads from other processes.

**value** (Input) The value used to initialize the value of the semaphore.

**attr** (Input) Attributes for the semaphore.

The members of the `sem_attr_np_t` structure are as follows.

<code>unsigned int reserved1[1]</code>	A reserved field that must be set to zero.
<code>unsigned int maxvalue</code>	The maximum value that the semaphore may obtain. <i>maxvalue</i> must be greater than zero. If a <code>sem_post()</code> or <code>sem_post_np()</code> operation would cause the value of a semaphore to exceed its maximum value, the operation will fail, returning EINVAL.
<code>unsigned int reserved2[2]</code>	A reserved field that must be set to zero.
<code>char title[16]</code>	The title of the semaphore. The title is a null-terminated string that contains up to 16 bytes. Any bytes after the null character are ignored. The title is retrieved using the Open List of Interprocess Communication Objects (QP0ZOLIP) API.
<code>void * reserved3[2]</code>	A reserved field that must be set to zero.

## Authorities

None

## Return Value

0	<code>sem_init_np()</code> was successful.
-1	<code>sem_init_np()</code> was not successful. The <i>errno</i> variable is set to indicate the error.

## Error Conditions

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

### [EINVAL]

The value specified for the argument is not correct.

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

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

The *value* parameter is greater than the *maxvalue* field of the *attr* parameter.

The *maxvalue* field of the *attr* parameter is greater than SEM\_VALUE\_MAX.

The *maxvalue* field of the *attr* parameter is equal to zero.

The reserved fields of the *attr* argument are not set to zero.

#### [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.

#### [ENOSPC]

No space available.

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

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

System semaphore resources have been exhausted.

## Error Messages

None.

## Related Information

- The `<semaphore.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`sem_destroy()`—Destroy Unnamed Semaphore” on page 70—Destroy Unnamed Semaphore
- “`sem_getvalue()`—Get Semaphore Value” on page 72—Get Semaphore Value
- “`sem_init()`—Initialize Unnamed Semaphore” on page 74—Initialize Unnamed Semaphore
- “`sem_post()`—Post to Semaphore” on page 85—Post to Semaphore
- “`sem_post_np()`—Post Value to Semaphore” on page 86—Post Value to Semaphore
- “`sem_trywait()`—Try to Decrement Semaphore” on page 89—Try to Decrement Semaphore
- “`sem_wait()`—Wait for Semaphore” on page 93—Wait for Semaphore
- “`sem_wait_np()`—Wait for Semaphore with Timeout” on page 95—Wait for Semaphore with Timeout

## Example

See Code disclaimer information for information pertaining to code examples.

The following example initializes an unnamed semaphore, `my_semaphore`, that will be used by threads of the current process and sets its value to 10. The maximum value and title of the semaphore are set to 10 and “MYSEM”.

```
#include <semaphore.h>
main() {
    sem_t my_semaphore;
    sem_attr_np_t attr;
    int rc;

    memset(&attr, 0, sizeof(attr));
    attr.maxvalue = 10;
    strcpy(attr.title, "MYSEM");
    rc = sem_init_np(&my_semaphore, 0, 10, &attr);
}
```

---

## sem\_open()—Open Named Semaphore

**Syntax**

```
#include <semaphore.h>
```

```
sem_t * sem_open(const char *name, int oflag, ...);
```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes

The **sem\_open()** function opens a named semaphore, returning a semaphore pointer that may be used on subsequent calls to **sem\_post()**, **sem\_post\_np()**, **sem\_wait()**, **sem\_wait\_np()**, **sem\_trywait()**, **sem\_getvalue()**, and **sem\_close()**. When a semaphore is being created, the parameters *mode* and *value* must be specified on the call to **sem\_open()**. If a semaphore is created, then the maximum value of the semaphore is set to SEM\_VALUE\_MAX and the title of the semaphore is set to the last 16 characters of the name.

If **sem\_open()** is called multiple times within the same process using the same name, **sem\_open()** will return a pointer to the same semaphore, as long as another process has not used **sem\_unlink()** to unlink the semaphore.

If **sem\_open()** is called from a program using data model LLP64, the returned semaphore pointer must be declared as a `sem_t *__ptr128`.

### Parameters

**name** (Input) A pointer to the null-terminated name of the semaphore to be opened. The name should begin with a slash (‘/’) character. If the name does not begin with a slash (‘/’) character, the system adds a slash to the beginning of the name.

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

The name is added to a set of names that is used only by named semaphores. The name has no relationship to any file system path names. The maximum length of the name is SEM\_NAME\_MAX.

See “QlgSem\_open()—Open Named Semaphore (using NLS-enabled path name)” on page 24—Open Named Semaphore (using NLS-enabled path name) for a description and an example of supplying the *name* in any CCSID.

**oflag** (Input) Option flags.

The *oflag* parameter value is either zero or is obtained by performing an OR operation on one or more of the following constants:

‘0x0008’ or O\_CREAT      Creates the named semaphore if it does not already exist.

'0x0010' or O\_EXCL Causes **sem\_open()** to fail if O\_CREAT is also set and the named semaphore already exists.

**mode** (input) Permission flags.

The *mode* parameter value is either zero or is obtained by performing an OR operation on one or more of the following list of constants. For another process to open the semaphore, the process's effective UIDd must be able to open the semaphore in both read and write mode.

'0x0100' or S\_IRUSR Permits the creator of the named semaphore to open the semaphore in read mode.  
'0x0080' or S\_IWUSR Permits the creator of the named semaphore to open the semaphore in write mode.  
'0x0020' or S\_IRGRP Permits the group associated with the named semaphore to open the semaphore in read mode.  
'0x0010' or S\_IWGRP Permits the group associated with the named semaphore to open the semaphore in write mode.  
'0x0004' or S\_IROTH Permits others to open the named semaphore in read mode.  
'0x0002' or S\_IWOTH Permits others to open the named semaphore in write mode.

**value** (Input) Initial value of the named semaphore.

## Authorities

### Authorization required for **sem\_open()**

Object Referred to	Authority Required	errno
Named semaphore to be created	None	None
Existing named semaphore to be accessed	*RW	EACCES

## Return Value

*value* **sem\_open()** was successful. The value returned is a pointer to the open named semaphore.  
SEM\_FAILED **sem\_open()** was not successful. The *errno* variable is set to indicate the error.

## Error Conditions

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

[EACCES]

Permission denied.

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

[EEXIST]

Semaphore exists.

A named semaphore exists for the parameter *name*, but O\_CREAT and O\_EXCL are both set in *oflag*.

[EINVAL]

The value specified for the argument is not correct.

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

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

The *value* parameter is greater than SEM\_VALUE\_MAX.

#### [ENAMETOOLONG]

The name is too long. The name is longer than the SEM\_NAME\_MAX characters.

#### [ENOENT]

No such path or directory.

The name specified on the **sem\_open()** call does not refer to an existing named semaphore and O\_CREAT was not set in *oflag*.

#### [ENOSPC]

No space available.

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

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

System semaphore resources have been exhausted.

## Error Messages

None.

## Related Information

- The <semaphore.h> file (see “Header Files for UNIX-Type Functions” on page 144)
- “QlgSem\_open()—Open Named Semaphore (using NLS-enabled path name)” on page 24 Open Named Semaphore (using NLS-enabled path name)
- “sem\_close()—Close Named Semaphore” on page 69—Close Named Semaphore
- “sem\_getvalue()—Get Semaphore Value” on page 72—Get Semaphore Value
- “sem\_open\_np()—Open Named Semaphore with Maximum Value” on page 81—Open Named Semaphore with Maximum Value
- “sem\_post()—Post to Semaphore” on page 85—Post to Semaphore
- “sem\_post\_np()—Post Value to Semaphore” on page 86—Post Value to Semaphore
- “sem\_trywait()—Try to Decrement Semaphore” on page 89—Try to Decrement Semaphore
- “sem\_unlink()—Unlink Named Semaphore” on page 91—Unlink Named Semaphore
- “sem\_wait()—Wait for Semaphore” on page 93—Wait for Semaphore
- “sem\_wait\_np()—Wait for Semaphore with Timeout” on page 95—Wait for Semaphore with Timeout

## Example

See Code disclaimer information for information pertaining to code examples.

The following example opens the named semaphore “/mysemaphore” and creates the semaphore with an initial value of 10 if it does not already exist. If the semaphore is created, the permissions are set such that only the current user has access to the semaphore.

```
#include <semaphore.h>
main() {
    sem_t * my_semaphore;
```



```

int rc;

my_semaphore = sem_open("/mysemaphore",
                        O_CREAT, S_IRUSR | S_IWUSR, 10);
}

```

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

---

## sem\_open\_np()—Open Named Semaphore with Maximum Value

### Syntax

```

#include <semaphore.h>

sem_t * sem_open_np(const char *name, int oflag,
                   mode_t mode, unsigned int value,
                   sem_attr_np_t * attr);

```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes

The **sem\_open\_np()** function opens a named semaphore, returning a semaphore pointer that may be used on subsequent calls to **sem\_post()**, **sem\_post\_np()**, **sem\_wait()**, **sem\_wait\_np()**, **sem\_trywait()**, **sem\_getvalue()**, and **sem\_close()**. If a named semaphore is being created, the parameters *mode*, *value*, and *attr* are used to set the permissions, value, and maximum value of the created semaphore.

If **sem\_open\_np()** is called multiple times within the same process using the same name, **sem\_open\_np()** will return a pointer to the same semaphore, as long as another process has not used **sem\_unlink()** to unlink the semaphore.

If **sem\_open\_np()** is called from a program using data model LLP64, the returned semaphore pointer must be declared as a `sem_t *__ptr128`.

## Parameters

**name** (Input) A pointer to the null-terminated name of the semaphore to be opened. The name should begin with a slash (‘/’) character. If the name does not begin with a slash (‘/’) character, the system adds a slash to the beginning of the name.

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

The name is added to a set of names used by named semaphores only. The name has no relationship to any file system path names. The maximum length of the name is SEM\_NAME\_MAX.

See “QlgSem\_open\_np()—Open Named Semaphore with Maximum Value (using NLS-enabled path name)” on page 26—Open Named Semaphore with Maximum Value (using NLS-enabled path name) for a description and an example of supplying the *name* in any CCSID.

**oflag** (Input) Option flags.

The *oflag* parameter value is either zero or is obtained by performing an OR operation on one or more of the following constants:

'0x0008' or  
O\_CREAT           Creates the named semaphore if it does not already exist.

'0x0010' or  
O\_EXCL           Causes **sem\_open\_np()** to fail if O\_CREAT is also set and the named semaphore already exists.

**mode** (input) Permission flags.

The *mode* parameter value is either zero or is obtained by performing an OR operation on one or more of the list of constants. For another process to open the semaphore, the process's effective UID must be able to open the semaphore in both read and write mode.

'0x0100' or  
S\_IRUSR           Permits the creator of the named semaphore to open the semaphore in read mode.

'0x0080' or  
S\_IWUSR           Permits the creator of the named semaphore to open the semaphore in write mode.

'0x0020' or  
S\_IRGRP           Permits the group associated with the named semaphore to open the semaphore in read mode.

'0x0010' or  
S\_IWGRP           Permits the group associated with the named semaphore to open the semaphore in write mode.

'0x0004' or  
S\_IROTH           Permits others to open the named semaphore in read mode.

'0x0002' or  
S\_IWOTH           Permits others to open the named semaphore in write mode.

**value** (Input) The initial value of the named semaphore.

**attr** (Input) Attributes for the semaphore.

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

*unsigned int reserved1[1]*   A reserved field that must be set to zero.

*unsigned int maxvalue*    The maximum value that the semaphore may obtain. *maxvalue* must be greater than zero. If a **sem\_post()** or **sem\_post\_np()** operation would cause the value of a semaphore to exceed its maximum value, the operation will fail, returning EINVAL.

*unsigned int reserved2[1]*   A reserved field that must be set to zero.

*char title[16]*           The title of the semaphore. The title is a null-terminated string that has a maximum length of 16 bytes. The string is associated with the semaphore. If the first byte is zero, then the system assigns a title to the semaphore that is based on the semaphore name. The title is retrieved using the Open List of Interprocess Communication Objects (QPOZOLIP) API.

*void \* reserved3[2]*    A reserved field that must be set to zero.

## Authorities

### Authorization required for `sem_open_np()`

Object Referred to	Authority Required	errno
Named semaphore to be created	None	None
Existing named semaphore to be accessed	*RW	EACCES

## Return Value

*value*                    **sem\_open\_np()** was successful. The value returned is a pointer to the opened named semaphore.  
*SEM\_FAILED*            **sem\_open\_np()** was not successful. The *errno* variable is set to indicate the error.

## Error Conditions

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

### [EACCES]

Permission denied.

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

### [EEXIST]

A named semaphore exists for the parameter *name*, but O\_CREAT and O\_EXCL are both set in *oflag*.

### [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 *maxvalue* field of the *attr* argument is greater than SEM\_VALUE\_MAX.

The *maxvalue* field of the *attr* argument is equal to zero.

The *value* argument is greater than the *maxvalue* field of the *attr* argument.

The reserved fields of the *attr* argument are not set to zero.

### [ENAMETOOLONG]

The name is too long. The name is longer than the SEM\_NAME\_MAX characters.

### [ENOENT]

No such path or directory.

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

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

The name specified on the **sem\_open\_np()** call does not refer to an existing named semaphore and O\_CREAT was not set in *oflag*.

### [ENOSPC]

No space available.

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

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

System semaphore resources have been exhausted.

## Error Messages

None.

## Related Information

- The <semaphore.h> file (see “Header Files for UNIX-Type Functions” on page 144)
- “QlgSem\_open\_np()—Open Named Semaphore with Maximum Value (using NLS-enabled path name)” on page 26—Open Named Semaphore with Maximum Value (using NLS-enabled path name)
- “sem\_close()—Close Named Semaphore” on page 69—Close Named Semaphore
- “sem\_getvalue()—Get Semaphore Value” on page 72—Get Semaphore Value
- “sem\_open()—Open Named Semaphore” on page 78—Open Named Semaphore
- “sem\_post()—Post to Semaphore” on page 85—Post to Semaphore
- “sem\_post\_np()—Post Value to Semaphore” on page 86—Post Value to Semaphore
- “sem\_trywait()—Try to Decrement Semaphore” on page 89—Try to Decrement Semaphore
- “sem\_unlink()—Unlink Named Semaphore” on page 91—Unlink Named Semaphore
- “sem\_wait()—Wait for Semaphore” on page 93—Wait for Semaphore
- “sem\_wait\_np()—Wait for Semaphore with Timeout” on page 95—Wait for Semaphore with Timeout>

## Example

See Code disclaimer information for information pertaining to code examples.

The following example opens the named semaphore “/mysemaphore” and creates the semaphore with an initial value of 10 and a maximum value of 11. The permissions are set such that only the current user has access to the semaphore.

```
#include <semaphore.h>
main() {
    sem_t * my_semaphore;
    int rc;
    sem_attr_np_t attr;

    memset(&attr, 0, sizeof(attr));
    attr.maxvalue=11;
    my_semaphore = sem_open_np("/mysemaphore",
                              0_CREAT|0_EXCL,
                              S_IRUSR | S_IWUSR,
                              10,
                              &attr);
}
```

API introduced: V4R4

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

---

## sem\_post()—Post to Semaphore

### Syntax

```
#include <semaphore.h>
```

```
int sem_post(sem_t * sem);
```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes

The **sem\_post()** function posts to a semaphore, incrementing its value by one. If the resulting value is greater than zero and if there is a thread waiting on the semaphore, the waiting thread decrements the semaphore value by one and continues running.

## Parameters

**sem** (Input) A pointer to an initialized unnamed semaphore or opened named semaphore.

## Authorities

None

## Return Value

0 **sem\_post()** was successful.

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

## Error Conditions

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

### [EINVAL]

The 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.

Posting to the semaphore would cause its value to exceed its maximum value. The maximum value is SEM\_VALUE\_MAX or was set using **sem\_open\_np()** or **sem\_init\_np()**.

## Error Messages

None.

## Related Information

- The <**semaphore.h**> file (see “Header Files for UNIX-Type Functions” on page 144)
- “**sem\_close()**—Close Named Semaphore” on page 69—Close Named Semaphore
- “**sem\_destroy()**—Destroy Unnamed Semaphore” on page 70—Destroy Unnamed Semaphore
- “**sem\_getvalue()**—Get Semaphore Value” on page 72—Get Semaphore Value

- “sem\_init()—Initialize Unnamed Semaphore” on page 74—Initialize Unnamed Semaphore
- “sem\_open()—Open Named Semaphore” on page 78—Open Named Semaphore
- “sem\_open\_np()—Open Named Semaphore with Maximum Value” on page 81—Open Named Semaphore with Maximum Value
- “sem\_post\_np()—Post Value to Semaphore”—Post Value to Semaphore
- “sem\_trywait()—Try to Decrement Semaphore” on page 89—Try to Decrement Semaphore
- “sem\_unlink()—Unlink Named Semaphore” on page 91—Unlink Named Semaphore
- “sem\_wait()—Wait for Semaphore” on page 93—Wait for Semaphore
- “sem\_wait\_np()—Wait for Semaphore with Timeout” on page 95—Wait for Semaphore with Timeout

## Example

See Code disclaimer information for information pertaining to code examples.

The following example initializes an unnamed semaphore and posts to it, incrementing its value by 1.

```
#include <stdio.h>
#include <semaphore.h>
main() {
    sem_t my_semaphore;
    int value;

    sem_init(&my_semaphore, 0, 10);
    sem_getvalue(&my_semaphore, &value);
    printf("The initial value of the semaphore is %d\n", value);
    sem_post(&my_semaphore);
    sem_getvalue(&my_semaphore, &value);
    printf("The value of the semaphore after the post is %d\n", value);
}
```

## Output:

```
The initial value of the semaphore is 10
The value of the semaphore after the post is 11
```

API introduced: V4R4

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

---

## sem\_post\_np()—Post Value to Semaphore

Syntax

```
#include <semaphore.h>

int sem_post_np(sem_t * sem,
                sem_post_options_np_t *options);
```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes

The `sem_post_np()` function posts to a semaphore, incrementing its value by the increment specified in the `options` parameter. If the resulting value is greater than zero and if there are threads waiting on the semaphore, the waiting threads decrement the semaphore and continue running.

## Parameters

**sem** (Input) A pointer to an initialized unnamed semaphore or opened named semaphore.

### options

(Input) Post options.

The members of the `sem_post_options_np_t` structure are as follows.

*unsigned int reserved1[1]*

A reserved field that must be set to zero.

*unsigned int increment*

The value, greater than zero, used to increment the semaphore. If the value specified causes the value of a semaphore to exceed its maximum value, `sem_post_np()` will fail by returning [EINVAL].

*unsigned int reserved2[2]*

A reserved field that must be set to zero.

## Authorities

None

## Return Value

0

`sem_post_np()` was successful.

-1

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

## Error Conditions

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

[EINVAL]

The 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.

[EOVERFLOW]

Maximum value exceeded.

Posting to the semaphore would cause its value to exceed its maximum value. The maximum value is `SEM_VALUE_MAX` or was set using `sem_open_np()` or `sem_init_np()`.

The reserved fields of the `attr` argument are not set to zero.

## Error Messages

None.

## Related Information

- The `<semaphore.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`sem_close()`—Close Named Semaphore” on page 69—Close Named Semaphore

- “sem\_destroy()—Destroy Unnamed Semaphore” on page 70—Destroy Unnamed Semaphore
- “sem\_getvalue()—Get Semaphore Value” on page 72—Get Semaphore Value
- “sem\_init()—Initialize Unnamed Semaphore” on page 74—Initialize Unnamed Semaphore
- “sem\_init\_np()—Initialize Unnamed Semaphore with Maximum Value” on page 75—Initialize Unnamed Semaphore with Maximum Value
- “sem\_open()—Open Named Semaphore” on page 78—Open Named Semaphore
- “sem\_open\_np()—Open Named Semaphore with Maximum Value” on page 81—Open Named Semaphore with Maximum Value
- “sem\_post()—Post to Semaphore” on page 85—Post to Semaphore
- “sem\_trywait()—Try to Decrement Semaphore” on page 89—Try to Decrement Semaphore
- “sem\_unlink()—Unlink Named Semaphore” on page 91—Unlink Named Semaphore
- “sem\_wait()—Wait for Semaphore” on page 93—Wait for Semaphore
- “sem\_wait\_np()—Wait for Semaphore with Timeout” on page 95—Wait for Semaphore with Timeout

## Example

See Code disclaimer information for information pertaining to code examples.

The following example initializes an unnamed semaphore and posts to it, incrementing its value by 2.

```
#include <stdio.h>
#include <semaphore.h>
main() {
    sem_t my_semaphore;
    sem_post_options_np_t options;
    int value;

    sem_init(&my_semaphore, 0, 10);
    sem_getvalue(&my_semaphore, &value);
    printf("The initial value of the semaphore is %d.\n", value);
    memset(&options, 0, sizeof(options));
    options.increment=2;
    sem_post_np(&my_semaphore,&options);
    sem_getvalue(&my_semaphore, &value);
    printf("The value of the semaphore after the post is %d.\n", value);
}
```

## Output:

The initial value of the semaphore is 10.  
The value of the semaphore after the post is 12.

API introduced: V4R4

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



---

## sem\_trywait()—Try to Decrement Semaphore

### Syntax

```
#include <semaphore.h>
```

```
int sem_trywait(sem_t * sem);
```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes

The **sem\_trywait()** function attempts to decrement the value of the semaphore. The semaphore will be decremented if its value is greater than zero. If the value of the semaphore is zero, then **sem\_trywait()** will return -1 and set *errno* to EAGAIN.

### Parameters

**sem** (Input) A pointer to an initialized unnamed semaphore or opened named semaphore.

### Authorities

None

### Return Value

0                    **sem\_trywait()** was successful.  
-1                   **sem\_trywait()** was not successful. The *errno* variable is set to indicate the error.

### Error Conditions

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

#### [EAGAIN]

Operation would have caused the process to be suspended.

The value of the semaphore is currently zero and cannot be decremented.

#### [EINTR]

Interrupted function call.

#### [EINVAL]

The value specified for the argument is not correct.

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

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

### Error Messages

None.

## Related Information

- The `<semaphore.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`sem_close()`—Close Named Semaphore” on page 69—Close Named Semaphore
- “`sem_destroy()`—Destroy Unnamed Semaphore” on page 70—Destroy Unnamed Semaphore
- “`sem_getvalue()`—Get Semaphore Value” on page 72—Get Semaphore Value
- “`sem_init()`—Initialize Unnamed Semaphore” on page 74—Initialize Unnamed Semaphore
- “`sem_init_np()`—Initialize Unnamed Semaphore with Maximum Value” on page 75—Initialize Unnamed Semaphore with Maximum Value
- “`sem_open()`—Open Named Semaphore” on page 78—Open Named Semaphore
- “`sem_open_np()`—Open Named Semaphore with Maximum Value” on page 81—Open Named Semaphore with Maximum Value
- “`sem_post()`—Post to Semaphore” on page 85—Post to Semaphore
- “`sem_post_np()`—Post Value to Semaphore” on page 86—Post Value to Semaphore
- “`sem_unlink()`—Unlink Named Semaphore” on page 91—Unlink Named Semaphore
- “`sem_wait()`—Wait for Semaphore” on page 93—Wait for Semaphore
- “`sem_wait_np()`—Wait for Semaphore with Timeout” on page 95—Wait for Semaphore with Timeout

## Example

See Code disclaimer information for information pertaining to code examples.

The following example attempts to decrement a semaphore with a current value of zero.

```
#include <stdio.h>
#include <errno.h>
#include <semaphore.h>
main() {
    sem_t my_semaphore;
    int value;
    int rc;

    sem_init(&my_semaphore, 0, 1);
    sem_getvalue(&my_semaphore, &value);
    printf("The initial value of the semaphore is %d\n", value);
    sem_wait(&my_semaphore);
    sem_getvalue(&my_semaphore, &value);
    printf("The value of the semaphore after the wait is %d\n", value);
    rc = sem_trywait(&my_semaphore);
    if ((rc == -1) && (errno == EAGAIN)) {
        printf("sem_trywait did not decrement the semaphore\n");
    }
}
```

## Output:

```
The initial value of the semaphore is 1
The value of the semaphore after the wait is 0
sem_trywait did not decrement the semaphore
```

API introduced: V4R4

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

## sem\_unlink()—Unlink Named Semaphore

### Syntax

```
#include <semaphore.h>
```

```
int sem_unlink(const char *name);
```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes

The **sem\_unlink()** function unlinks a named semaphore. The name of the semaphore is removed from the set of names used by named semaphores. If the semaphore is still in use, the semaphore is not deleted until all processes using the semaphore have ended or have called **sem\_close()**. Using the name of an unlinked semaphore in subsequent calls to **sem\_open()** or **sem\_open\_np()** will result in the creation of a new semaphore with the same name if the **O\_CREAT** flag of the *oflag* parameter has been set.

## Parameters

**name** (Input) A pointer to the null-terminated name of the semaphore to be unlinked. The name should begin with a slash (/) character. If the name does not begin with a slash (/) character, the system adds a slash to the beginning of the name.

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

The name is present in a set of names used only by named semaphores. The name has no relation to any file system path names. The maximum length of the name is **SEM\_NAME\_MAX**.

See “QlgSem\_unlink()—Unlink Named Semaphore (using NLS-enabled path name)” on page 28—Unlink Named Semaphore (using NLS-enabled path name) for a description and an example of supplying the *name* in any CCSID.

## Authorities

### Authorization required for sem\_unlink()

Object Referred to	Authority Required	errno
Named semaphore to be deleted	See note	EACCES

**Note:** To unlink a named semaphore, the effective UID of the process must be the creator of the semaphore or the process must have \*ALLOBJ authority.

## Return Value

0 **sem\_unlink()** was successful.

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

## Error Conditions

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

### [EACCES]

Permission denied.

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

### [ENOENT]

No such path or directory.

The specified name does not refer to an existing named semaphore.

### [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.

### [ENAMETOOLONG]

The name is too long. The name is longer than the `SEM_NAME_MAX` characters.

## Error Messages

None.

## Related Information

- The `<semaphore.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`QlgSem_unlink()`—Unlink Named Semaphore (using NLS-enabled path name)” on page 28—Unlink Named Semaphore (using NLS-enabled path name)
- “`sem_open()`—Open Named Semaphore” on page 78—Open Named Semaphore
- “`sem_open_np()`—Open Named Semaphore with Maximum Value” on page 81—Open Named Semaphore with Maximum Value

## Example

See Code disclaimer information for information pertaining to code examples.

The following example unlinks the named semaphore `"/mysem"`.

```
#include <semaphore.h>
main() {
    int rc;

    rc = sem_unlink("/mysem");
}
```

API introduced: V4R4

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

---

## sem\_wait()—Wait for Semaphore

### Syntax

```
#include <semaphore.h>
```

```
int sem_wait(sem_t * sem);
```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes

The `sem_wait()` function decrements by one the value of the semaphore. The semaphore will be decremented when its value is greater than zero. If the value of the semaphore is zero, then the current thread will block until the semaphore's value becomes greater than zero.

## Parameters

**sem** (Input) A pointer to an initialized unnamed semaphore or opened named semaphore.

## Authorities

None

## Return Value

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

## Error Conditions

If `sem_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.

[EINTR]

Interrupted function call.

[EINVAL]

The value specified for the argument is not correct.

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

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

## Error Messages

None.

## Related Information

- The `<semaphore.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`sem_close()`—Close Named Semaphore” on page 69—Close Named Semaphore

- “sem\_destroy()—Destroy Unnamed Semaphore” on page 70—Destroy Unnamed Semaphore
- “sem\_getvalue()—Get Semaphore Value” on page 72—Get Semaphore Value
- “sem\_init()—Initialize Unnamed Semaphore” on page 74—Initialize Unnamed Semaphore
- “sem\_init\_np()—Initialize Unnamed Semaphore with Maximum Value” on page 75—Initialize Unnamed Semaphore with Maximum Value
- “sem\_open()—Open Named Semaphore” on page 78—Open Named Semaphore
- “sem\_open\_np()—Open Named Semaphore with Maximum Value” on page 81—Open Named Semaphore with Maximum Value
- “sem\_post()—Post to Semaphore” on page 85—Post to Semaphore
- “sem\_post\_np()—Post Value to Semaphore” on page 86—Post Value to Semaphore
- “sem\_trywait()—Try to Decrement Semaphore” on page 89—Try to Decrement Semaphore
- “sem\_unlink()—Unlink Named Semaphore” on page 91—Unlink Named Semaphore
- “sem\_wait\_np()—Wait for Semaphore with Timeout” on page 95—Wait for Semaphore with Timeout

## Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a semaphore with an initial value of 10. The value is decremented by calling `sem_wait()`.

```
#include <stdio.h>
#include <semaphore.h>
main() {
    sem_t my_semaphore;
    int value;

    sem_init(&my_semaphore, 0, 1);
    sem_getvalue(&my_semaphore, &value);
    printf("The initial value of the semaphore is %d\n", value);
    sem_wait(&my_semaphore);
    sem_getvalue(&my_semaphore, &value);
    printf("The value of the semaphore after the wait is %d\n", value);
}
```

## Output:

```
The initial value of the semaphore is 1
The value of the semaphore after the wait is 0
```

API introduced: V4R4

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

---

## sem\_wait\_np()—Wait for Semaphore with Timeout

### Syntax

```
#include <semaphore.h>
```

```
int sem_wait_np(sem_t * sem,  
                sem_wait_options_np_t * options);
```

Service Program Name: QP0ZPSEM

Default Public Authority: \*USE

Threadsafe: Yes

The **sem\_wait\_np()** function attempts to decrement by one the value of the semaphore. The semaphore will be decremented by one when its value is greater than zero. If the value of the semaphore is zero, then the current thread will block until the semaphore's value becomes greater than zero or until the timeout period specified on the options parameter has ended. If the semaphore is not decremented before the timeout ends, **sem\_wait\_np()** will return with an error, setting *errno* to [ETIMEDOUT].

## Parameters

**sem** (Input) A pointer to an initialized unnamed semaphore or opened named semaphore.

### options

(Input) A pointer to a semaphore wait (*sem\_wait\_options\_np\_t*) structure. The members of the *sem\_wait\_options\_np\_t* structure are as follows:

<i>unsigned int reserved1[2]</i>	A reserved field that must be set to zero.
<i>struct sem_timeout_t timeout</i>	The time, in MI time, that <b>sem_wait_np()</b> should wait for the semaphore. If the timeout is zero, <b>sem_wait_np()</b> will return immediately with <i>errno</i> set to [ETIMEDOUT] if the semaphore cannot be decremented. If a timeout value of 0xFFFFFFFF FFFFFFFF is specified, then <b>sem_wait_np()</b> will wait indefinitely. The maximum timeout that may be specified is 281 272 976 710 655 (2 ** 48 -1) microseconds. Any value larger than this, other than 0xFFFFFFFF FFFFFFFF, will cause <b>sem_wait_np()</b> to wait for the maximum timeout (281 272 976 710 655 microseconds). The <i>Qp0zCvtToMItime()</i> may be used to convert a <i>timeval</i> structure to the corresponding MI time.

## Authorities

None

## Return Value

0	<b>sem_wait_np()</b> was successful.
-1	<b>sem_wait_np()</b> was not successful. The <i>errno</i> variable is set to indicate the error.

## Error Conditions

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

[ECANCEL]

Operation canceled.

*[EDESTROYED]*

The semaphore was destroyed.

*[EINTR]*

Interrupted function call.

*[EINVAL]*

The value specified for the argument is not correct.

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

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

*[ETIMEDOUT]*

A remote host did not respond within the timeout period.

## Error Messages

None.

## Related Information

- The `<semaphore.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`sem_close()`—Close Named Semaphore” on page 69—Close Named Semaphore
- “`sem_destroy()`—Destroy Unnamed Semaphore” on page 70—Destroy Unnamed Semaphore
- “`sem_getvalue()`—Get Semaphore Value” on page 72—Get Semaphore Value
- “`sem_init()`—Initialize Unnamed Semaphore” on page 74—Initialize Unnamed Semaphore
- “`sem_init_np()`—Initialize Unnamed Semaphore with Maximum Value” on page 75—Initialize Unnamed Semaphore with Maximum Value
- “`sem_open()`—Open Named Semaphore” on page 78—Open Named Semaphore
- “`sem_open_np()`—Open Named Semaphore with Maximum Value” on page 81—Open Named Semaphore with Maximum Value
- “`sem_post()`—Post to Semaphore” on page 85—Post to Semaphore
- “`sem_post_np()`—Post Value to Semaphore” on page 86—Post Value to Semaphore
- “`sem_trywait()`—Try to Decrement Semaphore” on page 89—Try to Decrement Semaphore
- “`sem_unlink()`—Unlink Named Semaphore” on page 91—Unlink Named Semaphore
- “`sem_wait_np()`—Wait for Semaphore with Timeout” on page 95—Wait for Semaphore with Timeout

## Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a semaphore with an initial value of 1. The value is decremented using `sem_wait()`. The program then attempts to decrement the semaphore using `sem_wait_np()` with a timeout of 2 seconds. This will fail with `ETIMEDOUT` because the semaphore’s value is currently zero.

```
#include <stdio.h>
#include <errno.h>
#include <semaphore.h>
#include <time.h>
#include <qp0z1170.h>

main() {
    sem_t my_semaphore;
    int value;
    sem_wait_options_np_t options;
```



```

int rc;
struct timeval waittime;
time_t start_time;
time_t end_time;

sem_init(&my_semaphore, 0, 1);
sem_getvalue(&my_semaphore, &value);
printf("The initial value of the semaphore is %d\n", value);
sem_wait(&my_semaphore);
sem_getvalue(&my_semaphore, &value);
printf("The value of the semaphore after the wait is %d\n", value);
memset(&options, 0, sizeof(options));
waittime.tv_sec = 2;
waittime.tv_usec = 0;
Qp0zCvtToMITime((unsigned char *) &options.timeout,
                wait_time,
                QP0Z_CVTTIME_TO_OFFSET);
time(&start_time);
rc = sem_wait_np(&my_semaphore, &options);
time(&end_time);
if ((rc == -1) && (errno == ETIMEDOUT)) {
    printf("sem_wait_np timed out after %d seconds\n",
          end_time - start_time);
}
}

```

## Output:

```

The initial value of the semaphore is 1
The value of the semaphore after the wait is 0
sem_wait_np timed out after 2 seconds

```

API introduced: V4R4

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

---

## shmat()—Attach Shared Memory Segment to Current Process

### Syntax

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

```
void *shmat(int shmid, const void *shmaddr,
            int shmflg);
```

Service Program Name: QP0ZUSHR

Default Public Authority: \*USE

Threadsafe: Yes

The **shmat()** function attaches to the shared memory segment specified by *shmid* and returns the address of the shared memory segment.

The address specified by *shmaddr* is only used when **shmat()** is called from a program that uses data model \*LLP64 and attaches to a teraspace shared memory segment. Otherwise the address specified by *shmaddr* is ignored and the actual shared memory segment address is returned regardless of the value of *shmaddr*.

The system maintains status information about a shared memory segment which can be retrieved with the “shmctl()—Perform Shared Memory Control Operations” on page 101 function. When a shared memory segment is successfully attached, the system sets the members of the `shmid_ds` structure as follows:

- `shm_nattch` is incremented by 1.
- `shm_lpid` is set to the process ID of the calling thread.
- `shm_atime` is set to the current time.

## Parameters

**shmid** (Input) Shared memory identifier, a positive integer. It is returned by the “shmget()—Get ID of Shared Memory Segment with Key” on page 106 function and used to identify the shared memory segment.

### shmaddr

(Input) Shared memory address. The address at which the calling thread would like the shared memory segment attached.

### shmflg

(Input) Operations flags. The value of the `shmflg` parameter is either zero or is obtained by performing an OR operation on one or more of the following constants:

#### SHM\_RDONLY (0x00001000)

Attach the shared memory segment in read-only mode. This flag is valid only for teraspace shared memory segments.

## Authorities

### Authorization Required for shmat()

Object Referred to	Authority Required	errno
Shared memory segment to be attached in read/write memory	Read and Write	EACCES
Shared memory segment to be attached in read-only memory in a process's teraspace.	Read	EACCES

## Return Value

*value* **shmat()** was successful. The value returned is a pointer to the shared memory segment associated with the specified identifier.

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

## Error Conditions

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

### [EACCES]

Permission denied.

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

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

Operation permission is denied to the calling thread.

Shared memory operations are not permitted because the QSHRMEMCTL system value is set to 0.

The shared memory segment is to be attached in read/write mode and the calling thread does not read and write permission to the shared memory segment.

The shared memory segment is to be attached in read-only mode and the calling thread does not read permission to the shared memory segment.

#### [EADDRINUSE]

A damaged object was encountered.

Address already in use.

An attempt was made to attach to a teraspace shared memory segment with the SHM\_MAP\_FIXED\_NP attribute and the address range is not available in the teraspace of the current job.

#### [EDAMAGE]

A damaged object was encountered.

The shared memory segment has been damaged by a previous shared memory operation.

#### [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 *shmid* parameter is not a valid shared memory identifier.

#### [EOPNOTSUPP]

Operation not supported.

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

The SHM\_RDONLY flag is set in the *shmflg* parameter. Read-only shared memory segments are not supported for nonteraspace shared memory segments and for shared memory segments created with the SHM\_MAP\_FIXED\_NP attribute.

#### [ENOMEM]

Storage allocation request failed.

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

The available data space is not large enough to accommodate the shared memory segment.

#### [EUNKNOWN]

Unknown system state.

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

## Error Messages

None.

## Usage Notes

1. The only supported operation flag is SHM\_RDONLY. This operation flag is supported only when you attach to a teraspace shared memory segment. If *shmflg* specifies SHM\_RDONLY for a nonteraspace shared memory segment, then an [EOPNOTSUPP] error is returned. All other values for *shmflg* are ignored.
2. A module that was not created with teraspace memory enabled should not attach to a teraspace shared memory segment. The call to *shmat()* will succeed and return a pointer. Any attempt, however, by a module not created with teraspace memory enabled to use the returned pointer will result in an MCH3601 (Pointer not set for location referenced) exception.
3. When a job attaches to a shared memory segment that was created with the SHM\_MAP\_FIXED\_NP attribute, an address range within the job's teraspace is used for the shared memory mapping. When a subsequent job attaches to the shared memory segment, the same address range within its teraspace must be available. If the address range is not available, the call to *shmat()* will fail with an [EADDRINUSE] error.
4. The storage for a shared memory segment is allocated when the first job attaches to the shared memory segment. The storage is charged against the job's temporary storage limit. If the job does not have enough temporary storage to satisfy the request, the call to *shmat()* will fail with an [ENOMEM] error.

## Related Information

- The `<sys/shm.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “*shmat()*—Attach Shared Memory Segment to Current Process” on page 97—Perform Shared Memory Control Operations
- “*shmget()*—Get ID of Shared Memory Segment with Key” on page 106—Get ID of Shared Memory Segment with Key
- “*shmdt()*—Detach Shared Memory Segment from Calling Process” on page 104—Detach Shared Memory Segment from Calling Process

## Example

See Code disclaimer information for information pertaining to code examples.

For an example of using this function, see Using Semaphores and Shared Memory in Examples: APIs.

API introduced: V3R6

Top | UNIX-Type APIs | APIs by category

---

## shmctl()—Perform Shared Memory Control Operations

```
Syntax
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmids *buf);

Service Program Name: QP0ZUSHR

Default Public Authority: *USE

Threadsafe: Yes
```

The **shmctl()** function allows the caller to control the shared memory segment specified by the *shmid* parameter.

A shared memory segment is controlled by setting the *cmd* parameter to one of the following values:

### IPC\_RMID (0x00000000)

Remove the shared memory segment identifier *shmid* from the system and destroy the shared memory segment.

The IPC\_RMID command can be run only by a thread with appropriate privileges or one that has an effective user ID equal to the user ID of the owner or the user ID of the creator of the shared memory segment. The structure pointed to by *buf* is ignored and can be NULL.

### IPC\_SET (0x00000001)

Set the user ID of the owner, the group ID of the owner, the permissions, and the maximum number of bytes for the shared memory segment to the values in the *shm\_perm.uid*, *shm\_perm.gid*, and *shm\_perm.mode* members of the *shmids* data structure pointed to by *buf*.

The IPC\_SET command can be run only by a thread with appropriate privileges or one that has an effective user ID equal to the user ID of the owner or the user ID of the creator of the shared memory segment.

### IPC\_STAT (0x00000002)

Store the current value of each member of the *shmids* data structure into the structure pointed to by *buf*. The IPC\_STAT command requires read permission to the shared memory segment.

### SHM\_SIZE (0x00000006)

Set the size of the shared memory segment using the *shm\_segsz* member of the *shmids* data structure pointed to by *buf*. This value may be larger or smaller than the current size. This command is valid for nonteraspaces shared memory segments and for teraspaces shared memory segments created using the SHM\_RESIZE\_NP option of the **shmget()** function. The maximum size to which a nonteraspaces shared memory segment may be expanded is 16 773 120 bytes (16 MB minus 4096 bytes). The maximum size of a resizable teraspaces shared memory segment is 268 435 456 bytes (256 MB).

The SHM\_SIZE command can be run only by a thread with appropriate privileges or a thread that has an effective user ID equal to the user ID of the owner or the user ID of the creator of the shared memory segment.

If a shared memory segment is resized to a smaller size, other threads using the memory that is being removed from the shared memory segment may experience memory exceptions when accessing that memory.

## Parameters

**shmid** (Input) Shared memory identifier, a positive integer. It is returned by the “shmget()—Get ID of

Shared Memory Segment with Key” on page 106 function and used to identify the shared memory segment on which to perform the control operation.

**cmd** (Input) Command, the control operation to perform on the shared memory segment. Valid values are listed above.

**buf** (I/O) Pointer to the `shmid_ds` structure to be used to get or set shared memory information.

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

*struct ipc\_perm* The members of the `ipc_perm` structure are as follows:  
*shm\_perm*

- uid\_t uid*  
The user ID of the owner of the segment.
- gid\_t gid*  
The group ID of the owner of the segment.
- uid\_t cuid*  
The user ID of the creator of the segment.
- gid\_t cgid*  
The group ID of the creator of the segment.
- mode\_t mode*  
The permissions for the segment.
- size\_t shm\_segsz*  
The size of the segment in bytes.
- pid\_t shm\_lpid*  
The process ID of the last job to attach or detach to the segment using `shmat()` or `shmdt()`.
- pid\_t shm\_cpid*  
The process ID of the job that created the segment using `shmget()`.
- int shm\_nattch*  
The number of jobs attached to the segment.
- time\_t shm\_atime*  
The time the last job attached to the segment using `shmat()`.
- time\_t shm\_dtime*  
The time the last job detached from the segment using `shmdt()`.
- time\_t shm\_ctime*  
The time the last job changed the segment using `shmctl()`.

## Authorities

### Authorization Required for `shmctl()`

Object Referred to	Authority Required	errno
Shared memory segment for which state information is retrieved ( <i>cmd</i> = <code>IPC_STAT</code> )	Read	EACCES
Shared memory segment for which state information is set ( <i>cmd</i> = <code>IPC_SET</code> )	See Note	EPERM
Shared memory segment to be removed ( <i>cmd</i> = <code>IPC_RMID</code> )	See Note	EPERM
Shared memory segment to be resized ( <i>cmd</i> = <code>SHM_SIZE</code> )	See Note	EPERM

**Note:** To set shared memory segment information, to remove a shared memory segment, or to resize a shared memory segment, the thread must be the owner or creator of the shared memory segment or have appropriate privileges.

## Return Value

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

## Error Conditions

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

### [EACCES]

Permission denied.

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

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

The *cmd* parameter is `IPC_STAT` and the calling thread does not have read permission to shared memory segment.

### [EDAMAGE]

A damaged object was encountered.

The shared memory segment has been damaged by a previous shared memory operation.

### [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`.

One of the following has occurred:

- The *shmid* parameter is not a valid shared memory identifier.
- The *cmd* parameter is not a valid command.
- The *cmd* parameter is `SHM_SIZE`, and the teraspace shared memory segment cannot be resized because it was not created by specifying `SHM_RESIZE_NP` on the *shmflg* parameter of `shmget()`.
- The *cmd* parameter is `SHM_SIZE`, and the new size is not valid for the shared memory segment.

### [ENOMEM]

Storage allocation request failed.

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

A shared memory identifier segment is to be resized, but the amount of available physical memory is not sufficient to fulfill the request.

### [EPERM]

Operation not permitted.

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

The *cmd* parameter is `IPC_RMID` or `IPC_SET` and both of the following are true:

- the calling thread does not have the appropriate privileges.

- the effective user ID of the calling thread is not equal to the user ID of the owner or the user ID of the creator of the shared memory segment.

[EUNKNOWN]

Unknown system state.

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

## Error Messages

None.

## Usage Notes

1. “Appropriate privileges” is defined to be \*ALLOBJ special authority. If the user profile under which the thread is running does not have \*ALLOBJ special authority, the thread does not have appropriate privileges.

## Related Information

- The <sys/shm.h> file (see “Header Files for UNIX-Type Functions” on page 144)
- “shmat()—Attach Shared Memory Segment to Current Process” on page 97—Attach Shared Memory Segment to Current Process
- “shmdt()—Detach Shared Memory Segment from Calling Process”—Detach Shared Memory Segment from Calling Process
- “shmget()—Get ID of Shared Memory Segment with Key” on page 106—Get ID of Shared Memory Segment with Key

## Example

See Code disclaimer information for information pertaining to code examples.

For an example of using this function, see Using Semaphores and Shared Memory in Examples: APIs.

API introduced: V3R6

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

---

## shmdt()—Detach Shared Memory Segment from Calling Process

Syntax

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

```
int shmdt(const void *shmaddr);
```

Service Program Name: QP0ZUSHR

Default Public Authority: \*USE

Threadsafe: Yes

The **shmdt()** function detaches the shared memory segment specified by *shmaddr* from the calling job. The *shmaddr* is the value returned by the “shmat()—Attach Shared Memory Segment to Current Process” on page 97 function.



The system maintains status information about a shared memory segment which can be retrieved with the “shmctl()—Perform Shared Memory Control Operations” on page 101 function. When a shared memory segment is successfully detached, the system sets the members of the `shmid_ds` structure as follows:

- `shm_nattch` is decremented by 1.
- `shm_lpid` is set to the process ID of the calling thread.
- `shm_dtime` is set to the current time.

## Parameters

**shmaddr**

(Input) Address of the shared memory segment to be detached.

## Authorities

Authorization Required for `shmdt()`

Object Referred to	Authority Required	errno
Shared memory segment to be detached	None	None

## Return Value

- 0 `shmdt()` was successful.
- 1 `shmdt()` was not successful. The `errno` variable is set to indicate the error.

## Error Conditions

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

[EDAMAGE]

A damaged object was encountered.

The shared memory segment has been damaged by a previous shared memory operation.

[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 value of `shmaddr` is not the start address of a shared memory segment.

[ENOSYS]

Function not implemented.

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

The function is not implemented.

[EUNKNOWN]

Unknown system state.

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

## Error Messages

None.

## Usage Notes

1. This function does not delete the shared memory segment. To delete a shared memory segment, use the `shmctl()` function with the `cmd` parameter set to `IPC_RMID`.

## Related Information

- The `<sys/shm.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`shmat()`—Attach Shared Memory Segment to Current Process” on page 97—Attach Shared Memory Segment to Current Process
- “`shmctl()`—Perform Shared Memory Control Operations” on page 101—Perform Shared Memory Control Operations
- “`shmget()`—Get ID of Shared Memory Segment with Key”—Get ID of Shared Memory Segment with Key

## Example

See Code disclaimer information for information pertaining to code examples.

For an example of using this function, see Using Semaphores and Shared Memory in Examples: APIs.

API introduced: V3R6

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

---

## shmget()—Get ID of Shared Memory Segment with Key

Syntax

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

```
int shmget(key_t key, size_t size, int shmflg);
```

Service Program Name: QP0ZUSHR

Default Public Authority: \*USE

Threadsafe: Yes

The **shmget()** function either creates a new shared memory segment or returns the shared memory identifier associated with the *key* parameter for an existing shared memory segment. A new shared memory segment is created if one of the following is true:

- The *key* parameter is equal to `IPC_PRIVATE`.
- The *key* parameter does not already have a shared memory identifier associated with it and the `IPC_CREAT` flag is specified in the *shmflg* parameter.

The system maintains status information about a shared memory segment which can be retrieved with the “**shmctl()**—Perform Shared Memory Control Operations” on page 101 function. When a new shared memory segment is created, the system initializes the members of the `shmid_ds` structure as follows:

- `shm_perm.cuid` and `shm_perm.uid` are set equal to the effective user ID of the calling thread.
- `shm_perm.cgid` and `shm_perm.gid` are set equal to the effective group ID of the calling thread.
- The low-order 9 bits of `shm_perm.mode` are set equal to the low-order 9 bits of the *shmflg* parameter.
- `shm_segsz` is set to the value specified in the *size* parameter.
- `shm_ctime` is set to the current time.
- `shm_lpid`, `shm_nattch`, `shm_atime`, and `shm_dtime` are set to zero.

There are two types of shared memory segments: teraspace shared memory segments and nonteraspace shared memory segments. A teraspace shared memory segment is accessed by adding the shared memory segment to a job’s teraspace. A nonteraspace shared memory segment creates shared memory using OS/400 space objects.

Shared memory segments larger than 16 773 120 bytes (16 MB minus 4096 bytes) should be created as teraspace shared memory segments. The maximum size of a teraspace shared memory segment is 4 294 967 295 bytes (4GB - 1). The maximum size of a resizeable teraspace shared memory segment is 268 435 456 bytes (256 MB).

The maximum size of a nonteraspace shared memory segments is 16 776 960 bytes (16 MB - 256 bytes). When the operating system accesses a nonteraspace shared memory segment that has a size in the range 16 773 120 bytes (16 MB minus 4096 bytes) to 16 776 960 bytes (16 MB minus 256 bytes), a performance degradation will be observed.

The size of the shared memory segment can be changed after it is created using the “**shmctl()**—Perform Shared Memory Control Operations” on page 101 function. The size can only be changed if it is nonteraspace shared memory segment or if it is a teraspace shared memory segment and `SHM_RESIZE_NP` is specified in the *shmflg* parameter.

## Parameters

**key** (Input) The key associated with the shared memory segment. A key of `IPC_PRIVATE` (0x00000000) guarantees that a unique shared memory segment is created. A key can also be specified by the caller or generated by the “**ftok()**—Generate IPC Key from File Name” on page 3 function.

**size** (Input) The size of the shared memory segment being created. If an existing shared memory segment is being accessed, *size* may be zero.

### shmflg

(Input) Operation and permission flags. The value of the *shmflg* parameter is either zero or is obtained by performing an OR operation on one or more of the following constants:

#### **S\_IRUSR (0x00000100)**

Allow the owner of the shared memory segment to attach to it in read mode.

#### **S\_IWUSR (0x00000080)**

Allow the owner of the shared memory segment to attach to it in write mode.

**S\_IRGRP (0x00000020)**

Allow the group of the shared memory segment to attach to it in read mode.

**S\_IWGRP (0x00000010)**

Allow the group of the shared memory segment to attach to it in write mode.

**S\_IROTH (0x00000004)**

Allow others to attach to the shared memory segment in read mode.

**S\_IWOTH (0x00000002)**

Allow others to attach to the shared memory segment in write mode.

**IPC\_CREAT (0x00000200)**

Create the shared memory segment if it does not exist.

**IPC\_EXCL (0x00000400)**

Return an error if the IPC\_CREAT flag is set and the shared memory segment already exists.

**SHM\_TS\_NP (0x00010000)**

If creating a new shared memory segment, make the shared memory segment a teraspace shared memory segment. When a job attaches to the shared memory segment, the shared memory segment will be added to the job's teraspace. Some compilers permit the user to indicate that the teraspace versions of storage functions should be used. For example, if a C module is compiled using CRTCMOD TERASPACE(\*YES \*TSIFC), this flag will be set automatically.

If accessing an existing shared memory segment, only specify this constant if it was specified when the shared memory segment was created.

**SHM\_RESIZE\_NP (0x00040000)**

If creating a new teraspace shared memory segment, allow the size of the shared memory segment to be changed with the "shmctl()—Perform Shared Memory Control Operations" on page 101 function. The maximum size of this teraspace shared memory segment is 268 435 456 bytes (256 MB). This flag is ignored for nonteraspace shared memory segments. A nonteraspace shared memory segment may always be resized up to 16 773 120 bytes (16 MB - 4096 bytes).

**SHM\_MAP\_FIXED\_NP (0x00100000)**

If creating a new teraspace shared memory segment, make all jobs that successfully attach to the shared memory segment attach to the shared memory segment at the same address. The shared memory segment may not be attached in read-only mode. This flag is ignored for nonteraspace shared memory segments.

If accessing an existing shared memory segment, only specify this constant if it was specified when the shared memory segment was created.

## Authorities

### Authorization Required for shmget()

Object Referred to	Authority Required	errno
Shared memory segment to be created	None	None
Existing shared memory segment to be accessed	See Note	EACCES

**Note:** If the thread is accessing a shared memory segment that already exists, the mode specified in the last 9 bits of the *shmflg* parameter must be a subset of the mode of the existing shared memory segment.

## Return Value

*value*                **shmget()** was successful. The value returned is the shared memory identifier associated with the *key* parameter.

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

## Error Conditions

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

### [EACCES]

Permission denied.

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

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

A shared memory identifier exists for the parameter *key*, but permissions specified in the low-order 9 bits of *shmflg* are not a subset of the current permissions.

Shared memory operations are not permitted because the QSHRMEMCTL system value is set to 0.

### [EDAMAGE]

A damaged object was encountered.

The shared memory segment has been damaged by a previous shared memory operation.

### [EEXIST]

File exists.

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

The named file, directory, or path already exists.

A shared memory identifier exists for the *key* parameter and both the IPC\_CREAT and IPC\_EXCL flags are set in the *shmflg* parameter.

### [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.

One of the following has occurred:

- The value of the *size* parameter is less than the system-imposed minimum or greater than the system-imposed maximum.
- A shared memory identifier exists for the *key* parameter and the size of the segment associated with it is less than *size* and *size* is not zero.
- A shared memory identifier exists for the *key* parameter and the SHM\_MAP\_FIXED\_NP or SHM\_TS\_NP attributes of the shared memory segment do not match the flags set in the *shmflg* parameter.

### [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.

A shared memory identifier does not exist for the *key* parameter and the `IPC_CREAT` flag is not set in the *shmflg* parameter.

#### [ENOMEM]

Storage allocation request failed.

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

A new shared memory segment is being created and the amount of available physical memory is not sufficient to fulfill the request.

#### [ENOSPC]

No space available.

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

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

A shared memory identifier cannot be created because the system limit on the maximum number of allowed shared memory identifiers would be exceeded.

#### [EUNKNOWN]

Unknown system state.

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

## Error Messages

None.

## Usage Notes

1. The best way to generate a unique key is to use the “`ftok()`—Generate IPC Key from File Name” on page 3 function.
2. When the operating system accesses a nonteraspace shared memory segment that has a size in the range 16 773 120 bytes (16 MB minus 4096 bytes) to 16 776 960 bytes (16 MB minus 256 bytes), a performance degradation will be observed. Use a teraspace shared memory segment if the size of the segment is larger than 16 773 120 bytes.
3. Use the “`shmat()`—Attach Shared Memory Segment to Current Process” on page 97 function to get addressability to the shared memory segment after the shared memory identifier is obtained.
4. The storage for a shared memory segment is not allocated until it is attached to a job. A job will not be able to attach to a shared memory segment that is larger than the amount of storage available on the system.
5. Jobs cannot attach a nonteraspace shared memory segment in read-only or write-only mode. Consequently, permissions that specify read-only or write-only will always result in `shmat()` returning with a return value of -1 and *errno* set to `EOPNOTSUPP`. Jobs are permitted to attach a teraspace shared memory segment in read-only mode.

## Related Information

- The `<sys/shm.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- “`ftok()`—Generate IPC Key from File Name” on page 3—Generate IPC Key from File Name
- “`shmat()`—Attach Shared Memory Segment to Current Process” on page 97—Attach Shared Memory Segment to Current Process

- “shmctl()—Perform Shared Memory Control Operations” on page 101—Perform Shared Memory Control Operations
- “shmdt()—Detach Shared Memory Segment from Calling Process” on page 104—Detach Shared Memory Segment from Calling Process

## Example

See Code disclaimer information for information pertaining to code examples.

For an example of using this function, see Using Semaphores and Shared Memory in Examples: APIs.

API introduced: V3R6

Top | UNIX-Type APIs | APIs by category

---

## Exit Programs

These are the Exit Programs for this category. [»](#)

---

### Integrated File System Scan on Close Exit Program

Required Parameter Group:

1 Integrated file system close exit information

**Input** Char(\*)

2 Status information

**Output** Char(\*)

QSYSINC Member Name: QP0LSCAN

Exit Point Name: QIBM\_QP0L\_SCAN\_CLOSE

Exit Point Format Name: SCCL0100

The integrated file system scan on close exit program is called to do scan processing when an integrated file system object is closed under the following conditions.

The exit program will **not** be called if:

- No exit programs exist for this exit point.
- -or- the Scan file systems (QSCANFS) system value has \*NONE specified so that no file systems will be scanned.
- -or- the object was marked to not be scanned and a scan is not required because the object was restored.
- -or- the object being closed was opened for write access only.
- -or- the object is the storage which was allocated for Integrated xSeries servers to use as virtual disk drives for the xSeries servers. From the perspective of the iSeries server, virtual drives appear as byte stream files within the integrated file system.
- -or- the object is not being accessed from a file server, and the Scan file systems control (QSCANFSCTL) system value has \*FSVROONLY specified so that only file server accesses are scanned.
- -or- the object is in a \*TYPE1 directory.

If the previous conditions have been met, the exit program will be called if:

- The object has never been scanned.
- -or- the object's data has been modified since the last time it was scanned. Data modifications include writes, memory map writes, truncates or clears.
- -or- the CCSID of the object has been modified since the last time it was scanned.
- -or- the To CCSID specified on the open request associated with this close is different than the last two To CCSIDs that were specified and previously scanned for this object.
- -or- the object was opened in binary in association with this close request, and it has not previously been scanned in binary.
- -or- there have been updates to the scanning software and the object was not marked to be scanned only if the object changed. Updates to scanning software occur by either registering additional exit programs for the scan-related exit points, or by calling Change Scan Signature (QP0LCHSG) API to update the scan key signature associated with existing exit program scan keys.

**Note:** If there are multiple descriptors referencing the same open instance of the object, then the exit program will **only** be called for the close request on the last descriptor. Additionally, the From CCSID of the object will be the value it is at the point in time of the close operation while the To CCSID will be reflective of the value specified at open.

For more information on close processing, see `close()`—Close File or Socket Descriptor. For more information on the scan-related attributes which can be set for objects, see `Qp0LSetAttr()`—Set Attributes. For more information on the integrated file system scan processing and various options, see the Integrated file system information in the Files and file systems topic.

The exit point supports a maximum of 50 exit programs. For information about adding an exit program to an exit point, see the Registration Facility.

**Note:** If the integrated file system exit program returns any error messages or if any errors are received when attempting to call the exit program, the object will be treated as if the program was not called and the object was not scanned. Therefore, the close operation will continue unless the Scan file systems control (QSCANFCTL) system value has \*ERRFAIL specified which will cause the operation to fail. If a scan detects a failure, the close operation will still proceed and complete to release the resources. If the Scan file systems control (QSCANFCTL) system value has \*NOFAILCLO specified, the close operation will not return any failure indication. If \*NOFAILCLO is not specified, the close operation will fail with error code [ESCANFAILURE].

## Restrictions

- Only objects of type \*STMF that are in \*TYPE2 directories in the "root" (/), QOpenSys, and user-defined file systems are scanned. For information on \*TYPE2 directories, see the Convert Directory(CVTDIR) command and the Integrated file system information in the Files and file systems topic.
- The exit programs will not be called during an IPL or the vary-on of an independent Auxiliary Storage Pool (ASP).
- The exit programs will not be called when objects are being closed as a part of a process end request.
- During the call to the exit programs, the ASP group associated with the thread will not be able to be changed.
- The exit programs must exist in the system ASP or in a basic user ASP. They cannot exist in an independent ASP. Any ASP group could be associated with the thread when the exit program is called. If the exit program is not found, the object will be treated as if the program was not called and the object was not scanned. Therefore, the close operation will continue unless the Scan file systems control (QSCANFCTL) system value has \*ERRFAIL specified which will cause the operation to fail. If the



Scan file systems control (QSCANFCTL) system value has \*NOFAILCLO specified, the close operation will not return any failure indication. If \*NOFAILCLO is not specified, the close operation will fail with error code [ESCANFAILURE].

- The exit programs could be called from an exit point within a multi-threaded job and must be written to be threadsafe.

## Authorities and Locks

### User Profile Authority

\*ALLOBJ (all object) and \*SECADM (security administrator) special authorities to add exit programs to the registration facility

\*ALLOBJ and \*SECADM special authorities to remove exit programs from the registration facility

## Program Data

When you register the exit program, the following program data must be provided. The following table shows the structure of the program data information. For a description of the fields in this format, see “Field Descriptions” on page 114. This structure is defined in header file qp0lscan.h as data type Qp0l\_Scan\_Program\_Data\_t.

Offset		Type	Field
Dec	Hex		
0	0	Char(10)	User profile
10	A	Char(20)	Scan key
30	1E	Char(12)	Scan key signature

## Required Parameter Group

### Integrated file system close exit information

INPUT; CHAR(\*)

Information that is needed by the exit program to do its object scan processing. For details, see “Format of Integrated File System Close Exit Information (Input).”

### Status information

OUTPUT; CHAR(\*)

Information that is returned by the exit program indicating what scan processing has occurred. For details, see “Format of Status Information (Output)” on page 114.

## Format of Integrated File System Close Exit Information (Input)

The following table shows the structure of the integrated file system close exit information for exit point format SCCL0100. For a description of the fields in this format, see “Field Descriptions” on page 114. This structure is defined in header file qp0lscan.h as data type Qp0l\_Scan\_Exit\_Information\_t.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Integrated file system close exit information length
4	4	CHAR(20)	Exit point name
24	18	CHAR(8)	Exit point format name
32	20	BINARY(4)	Length of status information
32	20	BINARY(4)	Scan descriptor
36	24	BINARY(4), UNSIGNED	From CCSID

Offset		Type	Field
Dec	Hex		
40	28	BINARY(4), UNSIGNED	To CCSID
44	2C	BINARY(4), UNSIGNED	Last failure CCSID
48	30	BINARY(4)	Oflags
52	34	CHAR(16)	File ID
68	44	CHAR(10)	Object type
78	4E	CHAR(1)	File system
79	4F	CHAR(1)	Additional call
80	50	CHAR(1)	Object modified since last scan
81	51	CHAR(1)	Scan signatures different
82	52	CHAR(1)	Call after previous failure

## Format of Status Information (Output)

The following table shows the structure of the status information. For a description of the fields in this format, see “Field Descriptions.” This structure is defined in header file `qp0lscan.h` as data type `Qp0l_Scan_Status_Information_t`.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Close scan status information length
4	4	BINARY(4), UNSIGNED	Failing CCSID
8	8	CHAR(1)	Update object scan information
9	9	CHAR(1)	Scan status

## Field Descriptions

**Additional call.** Whether the exit program was called an additional time because another “Integrated File System Scan on Close Exit Program” on page 111 that was called has indicated the object was modified. See the *scan status* field for this modify indication. The possible values are:

`QP0L_SCAN_CALL_FIRST (x'00')`

The first call to the exit program.

`QP0L_SCAN_CALL_ADDL (x'01')`

An additional call to the exit program because another exit program has indicated the object was modified.

**Call after previous failure.** Whether the exit program was called after the object had previously been scanned and a failure detected. The possible values are:

`QP0L_SCAN_NO (x'00')`

This is not a call after a previous scan failure.

`QP0L_SCAN_YES (x'01')`

This is a call after a previous scan failure. The *Last failure CCSID* field in conjunction with the *From CCSID* indicate the CCSID or binary indication of the failing scan request.

**Note:** If the *Failing CCSID* and *From CCSID* values match, it is the same as if the object would have been opened in binary.

**Close scan status information length.** The length in bytes of all data returned from the integrated file system close exit program. The only valid value for this field is 10. If anything else is specified, the object

will be treated as if the program was not called and the object was not scanned. Therefore, the close operation will continue unless the Scan file systems control (QSCANFSCCTL) system value has \*ERRFAIL specified which will cause the operation to fail. If the Scan file systems control (QSCANFSCCTL) system value has \*NOFAILCLO specified, the close operation will not return any failure indication. If \*NOFAILCLO is not specified, the close operation will fail with error code [ESCANFAILURE].

**Exit point format name.** The format name for the integrated file system scan on close exit program. The possible format name follows:

`SCCL0100` The format name that is used while an object is being closed.

**Exit point name.** The name of the exit point that is calling the exit program.

**Failing CCSID.** This field only has meaning if the *Call after previous failure* field had a value of QP0L\_SCAN\_YES when the exit program was called, and if the *Update object scan information* field has a value of QP0L\_SCAN\_YES, and if the *Scan status* field has a value of QP0L\_SCAN\_FAILURE or QP0L\_SCAN\_FAIL\_WANT\_MODIFY. When the *Call after previous failure* had a value of QP0L\_SCAN\_YES, then the scan exit program should verify that the object does not have any problems when scanned using both the *To CCSID* and *Last failure CCSID* values. If either scan fails, then this field should be filled in with the failing CCSID which will be stored as part of the object scan information with the failure indication. If the value of this field does not match either of the two input CCSID fields, then the *To CCSID* value will be used. If more than one exit program indicates a failure, the failing CCSID value which will be preserved is from the last exit program which scanned the object and indicated a failure. For more information on CCSIDs and conversions, see “Coded Character Set Identifier (CCSID) Information” on page 129 in “Integrated File System Scan on Open Exit Program” on page 121.

**Note:** If the *Failing CCSID* and *From CCSID* values match, it is the same as if the object would have been opened in binary.

**File ID.** A unique identifier associated with the object that is being closed. A file ID can be used with `Qp0lGetPathFromFileID()`—Get Path Name of Object from Its File ID, to retrieve an object’s path name.

**File system.** The file system that the object being scanned is in. The possible value follows:

`QP0L_SCAN_ROOT_QOPENSYS_UDFS (x'00')` The object is in the “root” (/), QOpenSys, or a user-defined file system.

**From CCSID.** The CCSID value that the data is in on the system itself at the point in time of the close operation. Therefore, this will be the CCSID in which data is to be returned (when reading from the object using the *Scan descriptor*), or the CCSID in which data is being supplied (when writing to the object using the *Scan descriptor*). For more information on CCSIDs and conversions, see “Coded Character Set Identifier (CCSID) Information” on page 129 in “Integrated File System Scan on Open Exit Program” on page 121.

**Integrated file system close exit information length.** The length in bytes of all data passed to the integrated file system close exit program.

**Last failure CCSID.** The CCSID value that was specified when this object was last scanned and indicated a scan failure. This field only has meaning if the *Call after previous failure* field has a value of QP0L\_SCAN\_YES. Therefore, this would have been the CCSID in which data was to have been returned (when the user was to be reading from the object), or the CCSID in which data was to have been supplied (when the user was to be writing to the object). However, that request failed for this CCSID. This is now being returned so that this CCSID can also be scanned, if it is different than the *To CCSID* value. For more information on CCSIDs and conversions, see “Coded Character Set Identifier (CCSID) Information” on page 129 in “Integrated File System Scan on Open Exit Program” on page 121.

**Note:** If the *Last failure CCSID* and *From CCSID* values match, it is the same as if the object would have been opened in binary.

**Length of status information.** The length in bytes allocated for the returned status information.

**Object modified since last scan.** Whether the exit program was called because the objects data or CCSID has been modified since it was last scanned. Examples of object data or CCSID modifications are: writing to the object, directly or through memory mapping; truncating the object; clearing the object; and changing the objects CCSID attribute, etc.. The possible values are:

<i>QP0L_SCAN_NO</i> (x'00')	The object has not been modified since it was last scanned.
<i>QP0L_SCAN_YES</i> (x'01')	The object has been modified since it was last scanned.

**Object type.** The object type. See Control Language (CL) information in the iSeries Information Center for descriptions of all iSeries object types.

**Oflags.** The oflags that were specified on the open request associated with this close request with the following exceptions. For a description of all possible oflag values, see `open()`—Open a File.

- If the oflags do not contain write access, the system will attempt to upgrade the access intent to include write, unless the Scan file systems control (QSCANFCTL) system value has \*NOWRTUPG specified or the object is not eligible for write access. If the upgrade is not attempted or is unsuccessful, the access intent matches the users invocation. If it is successful, the write access intent is included in this oflag information. This upgrade would be useful if the exit program wanted to modify the object to correct any problems found while scanning.
- The CCSID related flags will have been removed. This includes `O_TEXTDATA`, `O_CCSID`, `O_CODEPAGE`, and `O_TEXT_CREATE`.
- The synchronization flags will have been removed. This includes `O_SYNC`, `O_DSYNC`, and `O_RSYNC`.

**Scan descriptor.** A descriptor representing the object that is being closed. This scan descriptor has the following characteristics:

- It can be used to do any read processing on the object being processed. Reads using this descriptor will not update the last access timestamp information for the object.
- It can be used to do any write processing on the object being processed. If write processing is done by the exit program, the exit program should indicate `QP0L_SCAN_MODIFY` in the *Scan status* field. If it does not, the object's scan information will be cleared as if the objects data has been modified.
- It cannot be used to memory map the object, see `mmap()`—Memory Map a File.
- It cannot be used to close the object using `close()`—Close File or Socket Descriptor. When control returns from the exit program, the system code will do the close of this *scan descriptor*. The system will wait on this close attempt until all accesses to this object are closed. Therefore, if the exit program uses `givedescriptor()`—Pass Descriptor Access to Another Job and `takedescriptor()`—Receive Socket Access from Another Job or `sendmsg()`—Send Data or Descriptors or Both and `recvmsg()`—Receive Data or Descriptors or Both to pass the descriptor to another job, the job which used `takedescriptor()` or `recvmsg()` must close that descriptor when it is done processing, else the system will be waiting for that close.
- `dup()`—Duplicate Open File Descriptor and `fcntl()`—Perform File Control Command with `F_DUPFD` cannot be used to duplicate the *scan descriptor*. This is so the system has tight control of the closing of this scan descriptor.
- Data read using this descriptor will be in the *From CCSID* format. If any data is written using this descriptor, it must be in the *From CCSID* format. For more information on CCSIDs see “Coded Character Set Identifier (CCSID) Information” on page 129 in “Integrated File System Scan on Open Exit Program” on page 121.
- It will be a different descriptor than was specified on the close request.
- The oflags for this descriptor are what are passed on this interface.

- It is scoped to the process. However, one can use `givedescriptor()` and `takedescriptor()` or `sendmsg()` and `recvmsg()` to pass this descriptor to another job or process. Again, that process must complete its use of that descriptor before control is returned to the system from the exit program because the system will close the descriptor when exit program processing is complete. The system will wait on this close attempt until all accesses to this object are closed.
- No other threads in the process, other than those created by the exit program, will be able to access this descriptor.
- It only lives for the life of the exit program invocation. That is, once control is returned from the exit program, it will be destroyed. Therefore, it cannot be stored for later use.

**Scan key.** The scan key associated with this exit program. The first character of this scan key can not be hex zeros or a blank. For more information on the scan key, see “Scan Key List and Scan Key Signatures” on page 128 in “Integrated File System Scan on Open Exit Program” on page 121.

**Scan key signature.** The scan key signature associated with the specified *scan key*. For more information on the scan key signature, see “Scan Key List and Scan Key Signatures” on page 128 in “Integrated File System Scan on Open Exit Program” on page 121. If the specified *scan key* already exists in the scan key list, and the exit program is being added to replace an existing exit program entry, then the specified *scan key signature* must match the scan key signature associated with the scan key in the scan key list. If the specified *scan key* already exists in the scan key list, and the exit program is **not** being added to replace an existing exit program entry, then the specified *scan key signature* must match the scan key signature associated with the scan key in the scan key list unless the scan key signature associated with the scan key in the scan key list is all hex zeros. More than one exit program, including exit programs associated with the “Integrated File System Scan on Open Exit Program” on page 121, can have the same scan key signature.

**Scan signatures different.** Whether the exit program was called because the object’s current scan key signature is different than the appropriate associated signature. When an object is in an independent ASP group, the object scan signature is compared to the associated independent ASP group scan signature. When an object is **not** in an independent ASP group, the object scan signature is compared to the global scan signature. The possible values are:

<code>QP0L_SCAN_NO (x'00')</code>	The compared signatures are not different.
<code>QP0L_SCAN_YES (x'01')</code>	The compared signatures are different.

**Scan status.** The status of the scan processing. This field is only used if the *Update object scan information* field value specifies a value of `QP0L_SCAN_YES`. The possible values are:

<code>QP0L_SCAN_SUCCESS (x'01')</code>	The object was scanned and has no failures. If this indicator is returned by all exit programs that were called, the object will be marked as scan successful, and the close operation completes with no errors.
<code>QP0L_SCAN_FAILURE (x'02')</code>	The object was scanned and has at least one failure. If this indicator is returned by at least one of the exit programs that was called, the object will be marked as scan failure, and the close operation fails. Additionally, only the CCSID or binary indication related to this failing request will be kept in the object scan information, and the rest of the historical CCSID or binary information will be cleared.

Once an object has been marked as a failure under this condition, it will not be scanned again until the object’s scan signature is different than the global scan key signature or independent ASP group scan key signature as appropriate. Therefore, subsequent requests to work with the object will fail with a scan failure indication. Examples of requests which will fail are opening the object, changing the CCSID of the object, copying the object etc..

QP0L\_SCAN\_FAIL\_WANT\_MODIFY (x'03')

The object was scanned and has at least one failure. However, the exit program wanted to modify the file to correct the failure, but could not because it did not have write access. If this indicator is returned by at least one of the exit programs that was called, the object will be marked as scan failure, and the close operation fails. Additionally, only the CCSID or binary indication related to this failing request will be kept in the object scan information, and the rest of the historical CCSID or binary information will be cleared.

Once an object has been marked as a failure under this condition, it will not be scanned again until the object's scan signature is different than the global scan key signature or independent ASP group scan key signature as appropriate, or if a subsequent access would allow write access to be given to the exit program. Therefore, subsequent requests to work with the object will fail with a scan failure indication. Examples of requests which will fail are opening the object, changing the CCSID of the object, copying the object etc..

QP0L\_SCAN\_MODIFY (x'04')

The object was scanned, one or more failures were found, but the object was modified to remove the failures. If this indicator is returned by at least one of the exit programs that was called, then any exit programs which have previously been called will be called one more time so that they can scan the modified object information. This second call is indicated by an *Additional call* field value. If after this additional call, no failures are found, the object will be marked as scan successful, and the close operation completes with no errors.

If a value other than the possible values is specified, the object will be treated as if the program was not called and the object was not scanned. Therefore, the close operation will continue unless the Scan file systems control (QSCANFCTL) system value has \*ERRFAIL specified which will cause the operation to fail. If the Scan file systems control (QSCANFCTL) system value has \*NOFAILCLO specified, the close operation will not return any failure indication. If \*NOFAILCLO is not specified, the close operation will fail with error code [ESCANFAILURE].

**To CCSID.** The CCSID value that was specified on the open request associated with this close request. Therefore, this will be the CCSID in which data was returned (when the user was reading from the object), or the CCSID in which data was be supplied (when the user was writing to the object). Therefore, the exit program should be converting the data to this CCSID since this is how the data was presented to the user after their open request completed. For more information on CCSIDs and conversions, see "Coded Character Set Identifier (CCSID) Information" on page 129 in "Integrated File System Scan on Open Exit Program" on page 121.

**Note:** If the *To CCSID* and *From CCSID* values match, it is the same as if the object was opened in binary.

**Update object scan information.** Whether the scan information associated with the object should be updated or not. The object scan information includes the following:

- Scan status for the object.
- Scan signature associated with the object scan status.
- The *To CCSID* value of the object which was scanned or if the object was scanned in binary.

**Note:** Actually, the last two *To CCSID* values which have been scanned will be maintained as well as a separate indication of binary scans.

The possible values are:

QP0L\_SCAN\_NO (x'00')

The object scan information should not be updated. This might be used when the object was not actually scanned by the exit program, perhaps because it did not need to be, or perhaps because a deferred scan was initiated.

QP0L\_SCAN\_YES (x'01')

The object scan information should be updated. When this value is set, then the values in the *Scan status* field and *Failing CCSID* are used. If at least one exit program specified this value, then the object scan information will be updated.

If a value other than the possible values is specified, a value of QP0L\_SCAN\_NO is assumed.

**User profile.** The exit program will be called under this user profile. Therefore, this user profile should have \*USE authority to the exit program, and \*EXECUTE authority to the exit program library. If the user profile is not valid or accessible at the time the exit program is called, the object will be treated as if the program was not called and the object was not scanned. Therefore, the close operation will continue unless the Scan file systems control (QSCANFCTL) system value has \*ERRFAIL specified which will cause the operation to fail. If the Scan file systems control (QSCANFCTL) system value has \*NOFAILCLO specified, the close operation will not return any failure indication. If \*NOFAILCLO is not specified, the close operation will fail with error code [ESCANFAILURE]. The first character of the user profile can not be hex zeros or a blank.

**Note:** The system will not do any additional verification that this specified profile has authority to the object for which the exit program is being called when that exit program is being called, even when the access levels for the object are upgraded to include write. By registering this exit program, you are indicating this is acceptable.

## Usage Notes

1. When the exit program is executing (including any created threads), if it does any operations on other objects which might normally trigger another call to a scan-related exit program, the scan-related exit program will **not** be called, and it will be treated as if no scanning occurred for the object. For example, if the exit program opens a separate object, that object will not be scanned as part of that open request, even if an exit program is registered to the QIBM\_QP0L\_SCAN\_OPEN exit point. If however, that object has previously failed a scan, then the operation will fail with error code [ESCANFAILURE].
2. When the exit program is executing (including any created threads), if it does any opens of other objects, then the descriptors which will be returned will come from the same table of descriptors that the *Scan descriptor* is derived from. Therefore, customer application code will not be impacted by 'regular' descriptors being used and possibly reaching an application specified limit on the number of descriptors which can be used. Additionally, the exit program will not be able to use any of the 'regular' descriptors when it or any of its created threads are executing. That is, it will not be able to access any objects which have been opened outside the scope of the exit program execution. Any attempts to do so will fail with error code [EBADF].
3. When the following APIs are called from the thread executing the exit program and any of its created threads, the table of *Scan descriptors*, will not be inherited by the spawned process.
  - spawn()—Spawn Process
  - spawnp()—Spawn Process with PathTherefore, when the following APIs are called from the thread executing the exit program and any of its created threads, the descriptors returned by these APIs will only work within the same process.
  - pipe()—Create Interprocess Channel
  - Qp0zPipe()—Create Interprocess Channel with Sockets()
  - socketpair()—Create a Pair of Sockets
4. When the exit programs are executing (including any created threads), signals are blocked from being delivered to a thread. When a signal is blocked, the signal-handling action associated with the signal is not taken. The signal remains pending until all exit programs have completed execution. For more information, see Signal concepts.
5. When the following APIs are called from the thread executing the exit program and any of its created threads, they will fail with the listed error code.

- DosSetFileLocks()—Lock and Unlock a Byte Range of an Open File— [ENOTSUP]
  - DosSetFileLocks64()—Lock and Unlock a Byte Range of an Open File (Large File Enabled)— [ENOTSUP]
  - fcntl()—Perform File Control Command with F\_SETLK, F\_SETLK64, F\_SETLKW or F\_SETLKW64 — [ENOTSUP]
  - DosSetRelMaxFH()—Change maximum number of file descriptors — [ERROR\_GEN\_FAILURE]
  - dup2()—Duplicate Open File Descriptor to Another Descriptor — [ENOTSUP]
  - takedescriptor()—Receive Socket Access from Another Job — [ENOTSUP]
6. Unpredictable results will occur if the select()—Wait for events on multiple sockets API and any of its associated type and macro definitions are used in the thread executing the exit program and any of its created threads. Therefore, these interfaces should not be used under these conditions.
  7. It is recommended that the exit program use the large-file enabled APIs such as lseek64()—Set File Read/Write Offset (Large File Enabled) to work with the *scan descriptor* as these APIs will work with any size object.
  8. If Kerberos is configured on the system, then the thread executing the exit program and any of its created threads will not be able to access objects in any file systems which use Kerberos for authentication. If they do, the operation will fail with error code [ENOTSUP]. E.g. the exit program cannot access objects in the QFileSvr.400 file system when Kerberos is configured.
  9. The exit program should not call the open or close API interfaces on the object represented by the *scan descriptor*. If this is done from the thread executing the exit program, then [EDEADLK] will be returned. If the object is opened or closed from any other process or thread, that process or thread will wait until this invocation's scan is completed.

## Related Information

- The <qp0lscan.h> file (see “Header Files for UNIX-Type Functions” on page 144)
- Change Scan Signature (QP0LCHSG) API
- “Integrated File System Scan on Open Exit Program” on page 121
- Qp0lGetAttr()—Get Attributes.
- Qp0lSetAttr()—Set Attributes.
- Retrieve Scan Signature (QP0LRTSG) API
- Retrieve System Values (QWCRSVAL) API

◀ Exit program introduced: V5R3

Top | Integrated File System APIs | APIs by category





## Integrated File System Scan on Open Exit Program

Required Parameter Group:

1 Integrated file system open exit information

**Input** Char(\*)

2 Status information

**Output** Char(\*)

QSYSINC Member Name: QP0LSCAN

Exit Point Name: QIBM\_QP0L\_SCAN\_OPEN

Exit Point Format Name: SCOP0100

The integrated file system scan on open exit program is called to do scan processing when an integrated file system object is opened under the following conditions.

The exit program will **not** be called if:

- No exit programs exist for this exit point.
- -or- the Scan file systems (QSCANFS) system value has \*NONE specified so that no file systems will be scanned.
- -or- the object was marked to not be scanned and a scan is not required because the object was restored.
- -or- the object is being opened for write access only.
- -or- the object is being truncated as part of the open request.
- -or- the object is the storage which was allocated for Integrated xSeries servers to use as virtual disk drives for the xSeries servers. From the perspective of the iSeries server, virtual drives appear as byte stream files within the integrated file system.
- -or- the object is not being accessed from a file server, and the Scan file systems control (QSCANFSCTL) system value has \*FSVROONLY specified so that only file server accesses are scanned.
- -or- the object is in a \*TYPE1 directory.

If the previous conditions have been met, the exit program will be called if:

- The object has never been scanned.
- -or- the object's data has been modified since the last time it was scanned. Data modifications include writes, memory map writes, truncates or clears.
- -or- the CCSID of the object has been modified since the last time it was scanned.
- -or- the To CCSID specified on the open request is different than the last two To CCSIDs that were specified and previously scanned for this object.
- -or- the object is being opened in binary, and it has not previously been scanned in binary.
- -or- there have been updates to the scanning software and the object was not marked to be scanned only if the object changed. Updates to scanning software occur by either registering additional exit programs for the scan-related exit points, or by calling Change Scan Signature (QP0LCHSG) API to update the scan key signature associated with existing exit program scan keys.

For more information on open processing, as well as CCSID values, see `open()`—Open File. For more information on the scan-related attributes which can be set for objects, see `Qp0LSetAttr()`—Set Attributes.

For more information on the integrated file system scan processing and various options, see the Integrated file system information in the Files and file systems topic

The exit point supports a maximum of 50 exit programs. For information about adding an exit program to an exit point, see the Registration Facility.

**Note:** If the integrated file system exit program returns any error messages or if any errors are received when attempting to call the exit program, the object will be treated as if the program was not called and the object was not scanned. Therefore, the open operation will continue unless the Scan file systems control (QSCANFCTL) system value has \*ERRFAIL specified which will cause the operation to fail.

## Restrictions

- Only objects of type \*STMF that are in \*TYPE2 directories in the "root" (/), QOpenSys, and user-defined file systems are scanned. For information on \*TYPE2 directories, see the Convert Directory(CVTDIR) command and the Integrated file system information in the Files and file systems topic.
- The exit programs will not be called during an IPL or the vary-on of an independent Auxiliary Storage Pool (ASP).
- During the call to the exit programs, the ASP group associated with the thread will not be able to be changed.
- The exit programs must exist in the system ASP or in a basic user ASP. They cannot exist in an independent ASP. Any ASP group could be associated with the thread when the exit program is called. If the exit program is not found, the object will be treated as if the program was not called and the object was not scanned. Therefore, the open operation will continue unless the Scan file systems control (QSCANFCTL) system value has \*ERRFAIL specified which will cause the operation to fail.
- The exit programs could be called from an exit point within a multi-threaded job and must be written to be threadsafe.

## Authorities and Locks

### User Profile Authority

\*ALLOBJ (all object) and \*SECADM (security administrator) special authorities to add exit programs to the registration facility

\*ALLOBJ and \*SECADM special authorities to remove exit programs from the registration facility

## Program Data

When you register the exit program, the following program data must be provided. The following table shows the structure of the program data information. For a description of the fields in this format, see "Field Descriptions" on page 124. This structure is defined in header file qp0lscan.h as data type Qp0l\_Scan\_Program\_Data\_t.

Offset		Type	Field
Dec	Hex		
0	0	Char(10)	User profile
10	A	Char(20)	Scan key
30	1E	Char(12)	Scan key signature

## Required Parameter Group

### Integrated file system open exit information

INPUT; CHAR(\*)

Information that is needed by the exit program to do its object scan processing. For details, see “Format of Integrated File System Open Exit Information (Input).”

**Status information**

OUTPUT; CHAR(\*)

Information that is returned by the exit program indicating what scan processing has occurred. For details, see “Format of Status Information (Output).”

**Format of Integrated File System Open Exit Information (Input)**

The following table shows the structure of the integrated file system open exit information for exit point format SCOP0100. For a description of the fields in this format, see “Field Descriptions” on page 124. This structure is defined in header file qp0lscan.h as data type Qp0l\_Scan\_Exit\_Information\_t.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Integrated file system open exit information length
4	4	CHAR(20)	Exit point name
24	18	CHAR(8)	Exit point format name
32	20	BINARY(4)	Length of status information
36	24	BINARY(4)	Scan descriptor
40	28	BINARY(4), UNSIGNED	From CCSID
44	2C	BINARY(4), UNSIGNED	To CCSID
48	30	BINARY(4), UNSIGNED	Last failure CCSID
52	34	BINARY(4)	Oflags
56	38	CHAR(16)	File ID
72	48	CHAR(10)	Object type
82	52	CHAR(1)	File system
83	53	CHAR(1)	Additional call
84	54	CHAR(1)	Object modified since last scan
85	55	CHAR(1)	Scan signatures different
86	56	CHAR(1)	Call after previous failure

**Format of Status Information (Output)**

The following table shows the structure of the status information. For a description of the fields in this format, see “Field Descriptions” on page 124. This structure is defined in header file qp0lscan.h as data type Qp0l\_Scan\_Status\_Information\_t.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Open scan status information length
4	4	BINARY(4), UNSIGNED	Failing CCSID
8	8	CHAR(1)	Update object scan information
9	9	CHAR(1)	Scan status

## Field Descriptions

**Additional call.** Whether the exit program was called an additional time because another “Integrated File System Scan on Open Exit Program” on page 121 that was called has indicated the object was modified. See the *scan status* field for this modify indication. The possible values are:

<i>QP0L_SCAN_CALL_FIRST</i> (x'00')	The first call to the exit program.
<i>QP0L_SCAN_CALL_ADDL</i> (x'01')	An additional call to the exit program because another exit program has indicated the object was modified.

**Call after previous failure.** Whether the exit program was called after the object had previously been scanned and a failure detected. The possible values are:

<i>QP0L_SCAN_NO</i> (x'00')	This is not a call after a previous scan failure.
<i>QP0L_SCAN_YES</i> (x'01')	This is a call after a previous scan failure. The <i>Last failure CCSID</i> field in conjunction with the <i>From CCSID</i> indicate the CCSID or binary indication of the failing scan request.

**Note:** If the *Failing CCSID* and *From CCSID* values match, it is the same as if the object would have been opened in binary.

**Exit point format name.** The format name for the integrated file system scan on open exit program. The possible format name follows:

<i>SCOP0100</i>	The format name that is used while an object is being opened.
-----------------	---------------------------------------------------------------

**Exit point name.** The name of the exit point that is calling the exit program.

**Failing CCSID.** This field only has meaning if the *Call after previous failure* field had a value of *QP0L\_SCAN\_YES* when the exit program was called, and if the *Update object scan information* field has a value of *QP0L\_SCAN\_YES*, and if the *Scan status* field has a value of *QP0L\_SCAN\_FAILURE* or *QP0L\_SCAN\_FAIL\_WANT\_MODIFY*. When the *Call after previous failure* had a value of *QP0L\_SCAN\_YES*, then the scan exit program should verify that the object does not have any problems when scanned using both the *To CCSID* and *Last failure CCSID* values. If either scan fails, then this field should be filled in with the failing CCSID which will be stored as part of the object scan information with the failure indication. If the value of this field does not match either of the two input CCSID fields, then the *To CCSID* value will be used. If more than one exit program indicates a failure, the failing CCSID value which will be preserved is from the last exit program which scanned the object and indicated a failure. For more information on CCSIDs and conversions, see “Coded Character Set Identifier (CCSID) Information” on page 129.

**Note:** If the *Failing CCSID* and *From CCSID* values match, it is the same as if the object would have been opened in binary.

**File ID.** A unique identifier associated with the object that is being opened. A file ID can be used with *Qp0lGetPathFromFileID()*—Get Path Name of Object from Its File ID, to retrieve an object’s path name.

**File system.** The file system that the object being scanned is in. The possible value follows:

<i>QP0L_SCAN_ROOT_QOPENSYS_UDFS</i>	The object is in the “root” (/), QOpenSys, or a user-defined file system.
-------------------------------------	---------------------------------------------------------------------------

**From CCSID.** The CCSID value that the data is in on the system itself. Therefore, this will be the CCSID in which data is to be returned (when reading from the object using the *Scan descriptor*), or the CCSID in

which data is being supplied (when writing to the object using the *Scan descriptor*). For more information on CCSIDs and conversions, see “Coded Character Set Identifier (CCSID) Information” on page 129.

**Integrated file system open exit information length.** The length in bytes of all data passed to the integrated file system open exit program.

**Last failure CCSID.** The CCSID value that was specified when this object was last scanned and indicated a scan failure. This field only has meaning if the *Call after previous failure* field has a value of QP0L\_SCAN\_YES. Therefore, this would have been the CCSID in which data was to have been returned (when the user was to be reading from the object), or the CCSID in which data was to have been supplied (when the user was to be writing to the object). However, that request failed for this CCSID. This is now being returned so that this CCSID can also be scanned, if it is different than the *To CCSID* value. For more information on CCSIDs and conversions, see “Coded Character Set Identifier (CCSID) Information” on page 129.

**Note:** If the *Last failure CCSID* and *From CCSID* values match, it is the same as if the object would have been opened in binary.

**Length of status information.** The length in bytes allocated for the returned status information.

**Object modified since last scan.** Whether the exit program was called because the objects data or CCSID has been modified since it was last scanned. Examples of object data or CCSID modifications are: writing to the object, directly or through memory mapping; truncating the object; clearing the object; and changing the objects CCSID attribute, etc.. The possible values are:

QP0L_SCAN_NO (x'00')	The object has not been modified since it was last scanned.
QP0L_SCAN_YES (x'01')	The object has been modified since it was last scanned.

**Object type.** The object type. See Control Language (CL) information in the iSeries Information Center for descriptions of all iSeries object types.

**Oflags.** The oflags that were specified on the open request with the following exceptions. For a description of all possible oflag values, see open()—Open a File.

- If the oflags do not contain write access, the system will attempt to upgrade the access intent to include write, unless the Scan file systems control (QSCANFSCCTL) system value has \*NOWRTUPG specified or the object is not eligible for write access. If the upgrade is not attempted or is unsuccessful, the access intent matches the users invocation. If it is successful, the write access intent is included in this oflag information. This upgrade would be useful if the exit program wanted to modify the object to correct any problems found while scanning.
- The CCSID related flags will have been removed. This includes O\_TEXTDATA, O\_CCSSID, O\_CODEPAGE, and O\_TEXT\_CREATE.
- The synchronization flags will have been removed. This includes O\_SYNC, O\_DSYNC, and O\_RSYNC.

**Open scan status information length.** The length in bytes of all data returned from the integrated file system open exit program. The only valid value for this field is 10. If anything else is specified, the object will be treated as if the program was not called and the object was not scanned. Therefore, the open operation will continue unless the Scan file systems control (QSCANFSCCTL) system value has \*ERRFAIL specified which will cause the operation to fail.

**Scan descriptor.** A descriptor representing the object that is being opened. This scan descriptor has the following characteristics:

- It can be used to do any read processing on the object being processed. Reads using this descriptor will not update the last access timestamp information for the object.

- It can be used to do any write processing on the object being processed. If write processing is done by the exit program, the exit program should indicate QPOL\_SCAN\_MODIFY in the *Scan status* field. If it does not, the object's scan information will be cleared as if the object's data has been modified.
- It cannot be used to memory map the object, see `mmap()`—Memory Map a File.
- It cannot be used to close the object using `close()`—Close File or Socket Descriptor. When control returns from the exit program, the system code will do the close of this *scan descriptor*. The system will wait on this close attempt until all accesses to this object are closed. Therefore, if the exit program uses `givedescriptor()`—Pass Descriptor Access to Another Job and `takedescriptor()`—Receive Socket Access from Another Job or `sendmsg()`—Send Data or Descriptors or Both and `recvmsg()`—Receive Data or Descriptors or Both to pass the descriptor to another job, the job which used `takedescriptor()` or `recvmsg()` must close that descriptor when it is done processing, else the system will be waiting for that close.
- `dup()`—Duplicate Open File Descriptor and `fcntl()`—Perform File Control Command with `F_DUPFD` cannot be used to duplicate the *scan descriptor*. This is so the system has tight control of the closing of this scan descriptor.
- Data read using this descriptor will be in the *From CCSID* format. If any data is written using this descriptor, it must be in the *From CCSID* format. For more information on CCSIDs see “Coded Character Set Identifier (CCSID) Information” on page 129.
- It will be a different descriptor than will actually be returned to the user, if the open is ultimately successful.
- The `oflags` for this descriptor are what are passed on this interface.
- It is scoped to the process. However, one can use `givedescriptor()` and `takedescriptor()` or `sendmsg()` and `recvmsg()` to pass this descriptor to another job or process. Again, that process must complete its use of that descriptor before control is returned to the system from the exit program because the system will close the descriptor when exit program processing is complete. The system will wait on this close attempt until all accesses to this object are closed.
- No other threads in the process, other than those created by the exit program, will be able to access this descriptor.
- It only lives for the life of the exit program invocation. That is, once control is returned from the exit program, it will be destroyed. Therefore, it cannot be stored for later use.

**Scan key.** The scan key associated with this exit program. The first character of this scan key can not be hex zeros or a blank. For more information on the scan key, see “Scan Key List and Scan Key Signatures” on page 128

**Scan key signature.** The scan key signature associated with the specified *scan key*. For more information on the scan key signature, see “Scan Key List and Scan Key Signatures” on page 128. If the specified *scan key* already exists in the scan key list, and the exit program is being added to replace an existing exit program entry, then the specified *scan key signature* must match the scan key signature associated with the scan key in the scan key list. If the specified *scan key* already exists in the scan key list, and the exit program is **not** being added to replace an existing exit program entry, then the specified *scan key signature* must match the scan key signature associated with the scan key in the scan key list unless the scan key signature associated with the scan key in the scan key list is all hex zeros. More than one exit program, including exit programs associated with the “Integrated File System Scan on Close Exit Program” on page 111, can have the same scan key signature.

**Scan signatures different.** Whether the exit program was called because the object's current scan key signature is different than the appropriate associated signature. When an object is in an independent ASP group, the object scan signature is compared to the associated independent ASP group scan signature. When an object is **not** in an independent ASP group, the object scan signature is compared to the global scan signature. The possible values are:

<code>QPOL_SCAN_NO (x'00')</code>	The compared signatures are not different.
<code>QPOL_SCAN_YES (x'01')</code>	The compared signatures are different.

**Scan status.** The status of the scan processing. This field is only used if the *Update object scan information* field value specifies a value of QP0L\_SCAN\_YES. The possible values are:

QP0L\_SCAN\_SUCCESS (x'01')

The object was scanned and has no failures. If this indicator is returned by all exit programs that were called, the object will be marked as scan successful, and the open operation completes with no errors.

QP0L\_SCAN\_FAILURE (x'02')

The object was scanned and has at least one failure. If this indicator is returned by at least one of the exit programs that was called, the object will be marked as scan failure, and the open operation fails. Additionally, only the CCSID or binary indication related to this failing request will be kept in the object scan information, and the rest of the historical CCSID or binary information will be cleared.

Once an object has been marked as a failure under this condition, it will not be scanned again until the object's scan signature is different than the global scan key signature or independent ASP group scan key signature as appropriate. Therefore, subsequent requests to work with the object will fail with a scan failure indication. Examples of requests which will fail are opening the object, changing the CCSID of the object, copying the object etc..

QP0L\_SCAN\_FAIL\_WANT\_MODIFY (x'03')

The object was scanned and has at least one failure. However, the exit program wanted to modify the file to correct the failure, but could not because it did not have write access. If this indicator is returned by at least one of the exit programs that was called, the object will be marked as scan failure, and the open operation fails. Additionally, only the CCSID or binary indication related to this failing request will be kept in the object scan information, and the rest of the historical CCSID or binary information will be cleared.

Once an object has been marked as a failure under this condition, it will not be scanned again until the object's scan signature is different than the global scan key signature or independent ASP group scan key signature as appropriate, or if a subsequent access would allow write access to be given to the exit program. Therefore, subsequent requests to work with the object will fail with a scan failure indication. Examples of requests which will fail are opening the object, changing the CCSID of the object, copying the object etc..

QP0L\_SCAN\_MODIFY (x'04')

The object was scanned, one or more failures were found, but the object was modified to remove the failures. If this indicator is returned by at least one of the exit programs that was called, then any exit programs which have previously been called will be called one more time so that they can scan the modified object information. This second call is indicated by an *Additional call* field value. If after this additional call, no failures are found, the object will be marked as scan successful, and the open operation completes with no errors.

If a value other than the possible values is specified, the object will be treated as if the program was not called and the object was not scanned. Therefore, the open operation will continue unless the Scan file systems control (QSCANFCTL) system value has \*ERRFAIL specified which will cause the operation to fail.

**To CCSID.** The CCSID value that was specified on the open request. Therefore, this will be the CCSID in which data will be returned (when the user will be reading from the object), or the CCSID in which data will be supplied (when the user will be writing to the object). Therefore, the exit program should be converting the data to this CCSID since this is how the data will be presented to the user if the open request completes successfully. For more information on CCSIDs and conversions, see "Coded Character Set Identifier (CCSID) Information" on page 129.

**Note:** If the *To CCSID* and *From CCSID* values match, it is the same as if the object will be opened in binary.

**Update object scan information.** Whether the scan information associated with the object should be updated or not. The object scan information includes the following:

- Scan status for the object.
- Scan signature associated with the object scan status.
- The *To CCSID* value of the object which was scanned or if the object was scanned in binary.

**Note:** Actually, the last two *To CCSID* values which have been scanned will be maintained as well as a separate indication of binary scans.

The possible values are:

*QP0L\_SCAN\_NO* (x'00')

The object scan information should not be updated. This might be used when the object was not actually scanned by the exit program, perhaps because it did not need to be, or perhaps because a deferred scan was initiated.

*QP0L\_SCAN\_YES* (x'01')

The object scan information should be updated. When this value is set, then the values in the *Scan status* field and *Failing CCSID* are used. If at least one exit program specified this value, then the object scan information will be updated.

If a value other than the possible values is specified, a value of *QP0L\_SCAN\_NO* is assumed.

**User profile.** The exit program will be called under this user profile. Therefore, this user profile should have *\*USE* authority to the exit program, and *\*EXECUTE* authority to the exit program library. If the user profile is not valid or accessible at the time the exit program is called, the object will be treated as if the program was not called and the object was not scanned. Therefore, the open operation will continue unless the Scan file systems control (*QSCANFSCCTL*) system value has *\*ERRFAIL* specified which will cause the operation to fail. The first character of the user profile can not be hex zeros or a blank.

**Note:** The system will not do any additional verification that this specified profile has authority to the object for which the exit program is being called when that exit program is being called, even when the access levels for the object are upgraded to include write. By registering this exit program, you are indicating this is acceptable.

## Scan Key List and Scan Key Signatures

A list of *scan keys* and associated *scan key signatures* will be used to help minimize unnecessary scan calls, while allowing users to ensure scans occur when needed. The *scan key* list and *scan key signature* will allow an association of scanning software level with the various scan-related exit programs (“Integrated File System Scan on Close Exit Program” on page 111 and “Integrated File System Scan on Open Exit Program” on page 121). Updates to this information will allow the system to increment its global scan signature field to reflect the software updates.

The system will maintain a global scan signature field and independent ASP group scan signature fields. Each integrated file system object which supports scanning will have an object scan signature field.

The global scan signature indicates the state or level of the scanning software. It will or will not be modified under the following rules:

- When the scan-related exit programs are added or registered, the user specifies a scan key and a scan key signature. These values are added to the scan key list. If the scan key has previously been specified, e.g. for a different exit program registration, then the global scan signature will only be incremented if the specified scan key signature is not hex zero. If the scan key has not previously been specified, and the scan key signature is not a hex zero value, the global scan signature will be incremented.



- By calling the Change Scan Signature (QP0LCHSG) API to specify that a new scan key signature be associated with a specific scan key. This will cause the system to update the scan key list and increment the current global scan signature value.
- When the scan-related exit programs are removed, the user specifies a scan key and a scan key signature. These values are removed from the scan key list if no other scan-related exit programs are registered that have that scan key. Removing entries from the scan key list does **not** update the global scan signature.

The independent ASP group scan signature indicates the state of the scanning software as well. Since it moves with the independent ASP group, it represents the state of the scanning code software in relationship to when and where that independent ASP group was varied on. The independent ASP group scan key list and independent ASP group scan signature will or will not be modified under the following rules:

- If the independent ASP group is available and online, the independent ASP group scan key list will be updated whenever the system scan key list is updated. Any changes to the independent ASP group scan key list will cause the independent ASP group scan signature to be incremented under the same rules as to when the global scan signature is updated.
- If the independent ASP group is varied on after any global scan key list changes, then when the first scannable integrated file system object on the independent ASP group is opened or its scan information is retrieved, the independent ASP group scan key list will be compared to the global scan key list.
  - If the global scan key list has more scan keys or different scan key signatures than the independent ASP group scan key list has, then the independent ASP group scan list will be updated to match. Additionally, the independent ASP group scan signature will be incremented.
  - If the global scan key list is a proper subset of the scan keys and scan key signatures in the independent ASP group scan key list, then the independent ASP group scan list will be updated to match. However, the independent ASP group scan signature will not be incremented. If the global scan key list exactly matches the scan keys and scan key signatures in the independent ASP group scan key list, then no changes are made.

It is highly recommended that the scanning software level of support which is indicated by scan keys and scan key signatures be maintained the same across all systems in the independent ASP Cluster group. See Cluster for more information.

When an object in an independent ASP group is about to be scanned or its scan information is retrieved, the object scan signature will be compared to the associated independent ASP group scan signature. When an object is **not** in an independent ASP group, the object scan signature will be compared to the global scan signature value.

When an object is successfully scanned, the object scan signature will be updated to match the global scan signature or independent ASP group scan signature when scanning was begun as appropriate. Other associated fields will be updated as well as described in *Update object scan information*.

## Coded Character Set Identifier (CCSID) Information

The CCSID values presented on this interface have the following meanings and inter-relationships. The *From CCSID* represents the value for the data that is stored in the object. Therefore, when discussing reading and writing in the *From CCSID* format, it means the data is read or written as is, no conversion occurs between what is given to or returned by the system, and the data in the object itself. The *scan descriptor* that is passed to the exit program is not an open instance which provides CCSID conversion. But, when the object is ultimately opened, the file descriptor that is returned will include conversion using the value in *To CCSID*. If the *To CCSID* and *From CCSID* values match, it is the same as if the object would have been opened in binary. If the object is not being opened in binary, the scan exit program should do its scanning using the *To CCSID* value, and can use the appropriate APIs to do the conversion. If the scan succeeds or fails, then the CCSID which is preserved with the scan status information is the *To CCSID*, except for the following case. If the *Call after previous failure* field has a value of

QP0L\_SCAN\_YES, and the value in the *Last Failure CCSID* is different than *To CCSID*, then the scan exit program should also scan the object data using the *Last Failure CCSID*. In this case, if the scan succeeds, then the CCSID which is preserved with the scan status information is the *To CCSID*. If the scan fails, then the CCSID which is preserved with the scan status information is the *Failing CCSID*.

For more information on CCSIDs and conversions, see `open()`—Open a File and Globalization topic.

## Usage Notes

1. When the exit program is executing (including any created threads), if it does any operations on other objects which might normally trigger another call to a scan-related exit program, the scan-related exit program will **not** be called, and it will be treated as if no scanning occurred for the object. For example, if the exit program opens a separate object, that object will not be scanned as part of that open request, even if an exit program is registered to the QIBM\_QP0L\_SCAN\_OPEN exit point. If however, that object has previously failed a scan, then the operation will fail with error code [ESCANFAILURE].
2. When the exit program is executing (including any created threads), if it does any opens of other objects, then the descriptors which will be returned will come from the same table of descriptors that the *Scan descriptor* is derived from. Therefore, customer application code will not be impacted by 'regular' descriptors being used and possibly reaching an application specified limit on the number of descriptors which can be used.

Additionally, the exit program will not be able to use any of the 'regular' descriptors when it or any of its created threads are executing. That is, it will not be able to access any objects which have been opened outside the scope of the exit program execution. Any attempts to do so will fail with error code [EBADF].

3. When the following APIs are called from the thread executing the exit program and any of its created threads, the table of *Scan descriptors*, will not be inherited by the spawned process.

- `spawn()`—Spawn Process
- `spawnp()`—Spawn Process with Path

Therefore, when the following APIs are called from the thread executing the exit program and any of its created threads, the descriptors returned by these APIs will only work within the same process.

- `pipe()`—Create Interprocess Channel
- `Qp0zPipe()`—Create Interprocess Channel with Sockets()
- `socketpair()`—Create a Pair of Sockets

4. When the exit programs are executing (including any created threads), signals are blocked from being delivered to a thread. When a signal is blocked, the signal-handling action associated with the signal is not taken. The signal remains pending until all exit programs have completed execution. For more information, see Signal concepts.
5. When the following APIs are called from the thread executing the exit program and any of its created threads, they will fail with the listed error code.
  - `DosSetFileLocks()`—Lock and Unlock a Byte Range of an Open File— [ENOTSUP]
  - `DosSetFileLocks64()`—Lock and Unlock a Byte Range of an Open File (Large File Enabled)— [ENOTSUP]
  - `fcntl()`—Perform File Control Command with F\_SETLTK, F\_SETLTK64, F\_SETLKW or F\_SETLKW64 — [ENOTSUP]
  - `DosSetRelMaxFH()`—Change maximum number of file descriptors — [ERROR\_GEN\_FAILURE]
  - `dup2()`—Duplicate Open File Descriptor to Another Descriptor — [ENOTSUP]
  - `takedescriptor()`—Receive Socket Access from Another Job — [ENOTSUP]
6. Unpredictable results will occur if the `select()`—Wait for events on multiple sockets API and any of its associated type and macro definitions are used in the thread executing the exit program and any of its created threads. Therefore, these interfaces should not be used under these conditions.

7. It is recommended that the exit program use the large-file enabled APIs such as `lseek64()`—Set File Read/Write Offset (Large File Enabled) to work with the *scan descriptor* as these APIs will work with any size object.
8. If Kerberos is configured on the system, then the thread executing the exit program and any of its created threads will not be able to access objects in any file systems which use Kerberos for authentication. If they do, the operation will fail with error code `[ENOTSUP]`. E.g. the exit program cannot access objects in the `QFileSvr.400` file system when Kerberos is configured.
9. The exit program should not call the open or close API interfaces on the object represented by the *scan descriptor*. If this is done from the thread executing the exit program, then `[EDEADLK]` will be returned. If the object is opened or closed from any other process or thread, that process or thread will wait until this invocation's scan is completed.

## Related Information

- The `<qp0lscan.h>` file (see “Header Files for UNIX-Type Functions” on page 144)
- Change Scan Signature (QP0LCHSG) API
- “Integrated File System Scan on Close Exit Program” on page 111
- `Qp0lGetAttr()`—Get Attributes.
- `Qp0lSetAttr()`—Set Attributes.
- Retrieve Scan Signature (QP0LRTSG) API
- Retrieve System Values (QWCRSVAL) API

◀ Exit program introduced: V5R3

Top | Integrated File System APIs | APIs by category

---

## Process a Path Name Exit Program

Required Parameter Group:	
1	Selection status pointer
<b>Input</b>	BINARY(4)
2	Error value pointer
<b>Input</b>	BINARY(4)
3	Return value pointer
<b>Output</b>	BINARY(4)
4	Object name pointer
<b>Input</b>	CHAR(*)
5	Function control block pointer
<b>Input</b>	CHAR(*)

The Process a Path Name exit program is a user-specified exit program that is called by the `Qp0lProcessSubtree()` function for each object in the API's search that meets the caller's selection criteria. This exit program can be either a procedure or program.

When the user exit program is given control, it can call other APIs, build lists or tables, or do other processing. Since the API passes the names of all the children objects to the user exit program before passing the name of the parent, the user exit program can also delete directories.

If the exit program encounters an error during processing, it returns a valid *errno* in the Return value pointer field, that **Qp0lProcessSubtree()** returns to its caller. When its processing is complete, the exit program return code is set to tell **Qp0lProcessSubtree()** to do one of the following:

- End processing.
- Continue processing by calling the exit program again with the next object from the same directory.
- Continue processing by calling the exit program again, but not with objects from the same directory. In this case, **Qp0lProcessSubtree()** moves to the next directory or object that meets the specified criteria and calls the exit program with it.

If **Qp0lProcessSubtree()** encounters any problems in resolving to a user exit program, **Qp0lProcessSubtree()** ends and returns to its caller. If **Qp0lProcessSubtree()** encounters any errors with any other parameters, it ends and returns control to its caller, after a call to the user exit program. This call allows the exit program to perform any desired cleanup before **Qp0lProcessSubtree()** ends. Use the *Err\_recovery\_action* parameter of **Qp0lProcessSubtree()** to set other conditions for calling or not calling the user exit program.

Storage referred to by the Selection status pointer, Error value pointer, Return value pointer, or the Object name pointer when the Process a Path Name exit program is called, are destroyed or reused when **Qp0lProcessSubtree()** regains control.

See **Qp0lProcessSubtree()**—Process a Path Name for more information.

## Parameters

### *Selection status pointer*

INPUT; BINARY(4)

A pointer to an unsigned integer. This pointer indicates whether **Qp0lProcessSubtree()** encountered any problems in processing. Valid values follow:

- |   |                                                                                                                                                                                                                                                                          |
|---|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0 | <b>QP0L_SELECT_OK</b> : Indicates to that no problems were encountered during the selection of the current object. The Error value pointer parameter is set to NULL.                                                                                                     |
| 1 | <b>QP0L_SELECT_DONE</b> : Indicates that the last object was processed and that this is the last call to the Process a Path Name exit program. The Object name pointer and the Error value pointer parameters are set to NULL.                                           |
| 2 | <b>QP0L_SELECT_NOT_OK</b> : Indicates that <b>Qp0lProcessSubtree()</b> has encountered an error but that the Process a Path Name exit program can decide if the operation should continue or end. The Error value pointer parameter points to a valid <i>errno</i> .     |
| 3 | <b>QP0L_SELECT_FAILED</b> : Indicates that <b>Qp0lProcessSubtree()</b> has encountered an unrecoverable error and that <b>Qp0lProcessSubtree()</b> will return to its caller when it regains control. The Error value pointer parameter points to a valid <i>errno</i> . |

### *Error value pointer*

INPUT; BINARY(4)

A pointer to a valid *errno* that describes any problems encountered by the API during the processing of the current object. Any valid *errno* can be passed in this field when this parameter is not NULL.

### *Return value pointer*

OUTPUT; BINARY(4)

A pointer to a value from the Process a Path Name exit program that instructs the API to continue or to end processing. Valid values follow.

- 0            **Process a Path Name exit program** was successful.
- 1          **Process a Path Name exit program** was successful. **Qp0lProcessSubtree()** should skip processing any remaining objects in this directory and move on to process objects in other directories.
- > 0 (an errno)    **Process a Path Name exit program** was not successful. **Qp0lProcessSubtree()** ends.

**Object name pointer**

INPUT; CHAR(\*)

A pointer to the path name structure that contains the fully qualified name of the object being processed by **Qp0lProcessSubtree()**. The Path\_Type flag defined in the qlg.h header file must be used to determine whether the Object name pointer contains a pointer or is a character string. This flag must also be used to determine whether the path name delimiter character is 1 or 2 characters long. Value values follow:

- 0            The path name is a character string, and the path name delimiter is 1 character long.
- 1            The path name is a pointer, and the path name delimiter character is 1 character long.
- 2            The path name is a character string, and the path name delimiter is 2 characters long.
- 3            The path name is a pointer, and the path name delimiter character is 2 characters long.

**Function control block pointer**

INPUT; CHAR(\*)

A pointer to the data that is passed to **Qp0lProcessSubtree()** on its call. **Qp0lProcessSubtree()** does not process the data that is referred to by this pointer, but passes this pointer as a parameter when it calls the exit program.

Exit program introduced: V4R3

Top | UNIX-Type APIs | APIs by category

## Save Storage Free Exit Program

Required Parameter Group:	
1	Path name pointers
<b>Input</b>	Char(*)
2	Return code pointer
<b>Output</b>	Binary(4)
3	Return value pointer
<b>Output</b>	Binary(4)
4	Function control block pointer
<b>Input</b>	Char(*)

The Save Storage Free exit program is a user-specified program that is called by **Qp0lSaveStgFree()** to save an OS/400 object of type \*STMF. This exit program can be either a procedure or program.

When the Save Storage Free exit program is given control, it should save the object so it can be dynamically retrieved at a later time. The \*STMF object is locked when the exit program is called to prevent changes to it until the storage free operation is complete. If the Save Storage Free exit program

ends unsuccessfully, it must return a valid *errno* in the storage pointed to by the return value pointer. **Qp0lSaveStgFree()** then passes this *errno* to its caller with a minus one return code.

Storage referred to by the path name pointers or the return code pointer when the Save Storage Free exit program is called is destroyed or reused when **Qp0lSaveStgFree()** regains control.

## Required Parameter Group

### Path names pointers

INPUT; CHAR(\*)

All of the path names to the \*STMF object being storage freed. There is one path name for each link to the object. These path names are in the Qlg\_Path\_Name\_T format and are in the UCS-2 CCSID. See Path name format for more information on this format. For information about UCS-2, see the Globalization topic.

Path Name Pointers			
Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Number of path names
4	4	CHAR(12)	Reserved
16	10	ARRAY(*)	Array of path name pointers

**Array of path name pointers.** Pointers to each path name that **Qp0lSaveStgFree()** found for the object identified by the path name on the call to **Qp0lSaveStgFree()**. Each path name is in the Qlg\_Path\_Name\_T format.

**Number of path names.** The total number of path names that **Qp0lSaveStgFree()** found for the object identified by the caller of **Qp0lSaveStgFree()**.

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

### Return code pointer

OUTPUT; BINARY(4)

A pointer to an indicator that is returned to indicate whether the exit program was successful or whether it failed. Valid values follow:

0	The Save Storage Free exit program was successful.
-1	The Save Storage Free exit program was not successful. The Return value pointer is set to indicate the error.

### Return value pointer

OUTPUT; BINARY(4)

A pointer to a valid *errno* that is returned from the exit program to identify the reason it was not successful.

### Function control block pointer

INPUT; CHAR(\*)

A pointer to the data that is passed to **Qp0lSaveStgFree()** on its call. **Qp0lSaveStgFree()** does not process the data that is referred to by this pointer, but passes this pointer as a parameter when it calls the exit program.

## Related Information

- Qp0lSaveStgFree()—Save Storage Free

---

## Concepts

These are the concepts for this category.

---

### Identifier Based Services

Although each IPC service provides a specific type of interprocess communication, the three identifier based services share many similarities. Each service defines a mechanism through which its communications take place. For message queues, that mechanism is a message queue; for semaphore sets, it is a semaphore set; and for shared memory, it is a shared memory segment. These mechanisms are identified by a unique positive integer called, respectively, a message queue identifier (*msqid*), a semaphore set identifier (*semid*), and a shared memory identifier (*shmid*).

**Note:** Throughout the Interprocess Communication APIs, the term thread is used extensively. This does not mean that IPC objects can be used only between threads within one process, but rather that authorization checks and waits occur for the calling thread within a process.

Associated with each identifier is a data structure that contains state information for the IPC mechanism, as well as ownership and permissions information. The ownership and permissions information is defined in a structure in the `<sys/ipc.h>` header file as follows:

```
typedef struct ipc_perm {
    uid_t    uid;    /* Owner's user ID        */
    gid_t    gid;    /* Owner's group ID       */
    uid_t    cuid;   /* Creator's user ID      */
    gid_t    cgid;   /* Creator's group ID     */
    mode_t   mode;   /* Access modes           */
} ipc_perm_t;
```

This structure is similar to a file permissions structure, and is initialized by the thread that creates the IPC mechanism. It is then checked by all subsequent IPC operations to determine if the requesting thread has the required permissions to perform the operation.

To get an identifier, a thread must either create a new IPC mechanism or access an existing mechanism. This is done through the `msgget()`, `semget()`, and `shmget()` functions. Each get operation takes as input a *key* parameter and returns an identifier. Each get operation also takes a *flag* parameter. This *flag* parameter contains the IPC permissions for the mechanism as well as bits that determine whether or not a new mechanism is created. The rules for whether a new mechanism is created or an existing one is referred to are as follows:

- Specifying a *key* of `IPC_PRIVATE` guarantees a new mechanism is created.
- Setting the `IPC_CREAT` bit in the *flag* parameter creates a new mechanism for the specified *key* if one does not already exist. If an existing mechanism is found, its identifier is returned.
- Setting both `IPC_CREAT` and `IPC_EXCL` creates a new mechanism for the specified *key* only if a mechanism does not already exist. If an existing mechanism is found, an error is returned.

When a message queue, semaphore set, or shared memory segment is created, the thread that creates it determines how it can be accessed. The thread does this by passing mode information in the low-order 9 bits of the *flag* parameter on the `msgget()`, `semget()`, or `shmget()` function call. This information is used to initialize the mode field in the `ipc_perm` structure. The values of the bits are given below in hexadecimal notation:

Bit	Meaning
X"0100"	Read by user
X"0080"	Write by user
X"0020"	Read by group
X"0010"	Write by group
X"0004"	Read by others
X"0002"	Write by others

Subsequent IPC operations do a permission test on the calling thread before allowing the thread to perform the requested operation. This permission test is done in one of three forms:

- For the **msgget()**, **semget()**, or **shmget()** calls that are accessing an existing IPC mechanism, the caller's *flag* parameter is checked to make sure it does not specify access bits that are not in the mode field of the existing IPC mechanism's `ipc_perm` structure. If the *flag* parameter does not contain any bits that are not in the mode field, permission is granted.
- For most of the other IPC APIs, the effective user ID and effective group ID of the thread are retrieved, and these values are compared with the data in the `ipc_perm` structure as follows:
  - If the effective user ID equals either the `uid` or the `cuid` field for the IPC mechanism, and if the appropriate access bit is on in the mode field (either Read by user or Write by user, depending on the operation being requested), permission is granted.
  - If the effective group ID equals either the `gid` or the `cgid` field for the IPC mechanism, and if the appropriate access bit is on in the mode field (either Read by group or Write by group), permission is granted.
  - If none of the above tests are true, and if the appropriate access bit is on for others (either Read by others or Write by others), permission is granted.
- For the **msgctl()**, **semctl()**, or **shmctl()** APIs, some values of the *cmd* parameter require the caller to be the owner or creator of the IPC object, or have appropriate privileges. The values of *cmd* that this rule applies to depends on the API. This is shown in the API descriptions for **msgctl()**, **semctl()**, and **shmctl()**.

## Message Queues

**Message queues** provide a form of message passing in which any process (given that it has the necessary permissions) can read a message from or write a message to any IPC message queue on the system. There are no requirements that a process be waiting to receive a message from a queue before another process sends one, or that a message exist on the queue before a process requests to receive one.

Every message on a queue has the following attributes:

- Message type
- Message length (length of data part of message)
- Message data (if length is greater than 0)

A thread gets a message queue identifier by calling the **msgget()** function. Depending on the *key* and *msgflg* parameters passed in, either a new message queue is created or an existing message queue is accessed. When a new message queue is created, a data structure is also created to contain information about the message queue. This structure is defined in the `<sys/msg.h>` header file as follows:

```
typedef struct msgqid_ds {
    struct ipc_perm  msg_perm;    /* Operation permission struct */
    msgqnum_t       msg_qnum;    /* # msgs currently on queue */
    msglen_t        msg_qbytes;  /* Max # bytes allowed on queue*/
    pid_t           msg_lspid;   /* Process ID of last msgsnd() */
    pid_t           msg_lrpid;   /* Process ID of last msgrcv() */
};
```



```

    time_t      msg_stime; /* Time of last msgsnd()      */
    time_t      msg_rtime; /* Time of last msgrcv()   */
    time_t      msg_ctime; /* Time of last change     */
} msqid_ds_t;

```

A thread puts a message on a message queue by calling the **msgsnd()** function. The following parameters are passed in:

- Message queue ID
- Pointer to a buffer containing the message type and message data
- Length of the message
- Flag that specifies whether or not the thread is willing to wait to send the message.

A thread gets a message from a message queue by calling the **msgrcv()** function. The following parameters are passed in:

- Message queue ID
- Pointer to a buffer in which to receive the message
- Length of the buffer
- Type of message
- Flag that specifies whether or not the thread is willing to wait and whether or not the thread is willing to truncate a message to receive it

A thread removes a message queue ID by calling the **msgctl()** function. The thread also can use the **msgctl()** function to change the data structure values associated with the message queue ID or to retrieve the data structure values associated with the message queue ID. The following parameters are passed in:

- Message queue ID
- Command the thread wants to perform (remove ID, set data structure values, receive data structure values)
- Pointer to a buffer from which to set data structure values or in which to receive data structure values

## Message Queue Differences and Restrictions

OS/400 message queues differ from the message queue definition in the Single UNIX Specification in the following ways:

- The maximum message size is 65535 bytes.
- The maximum number of bytes on a message queue is 16 777 216.
- The maximum number of message queues that can be created (system-wide) is 2 147 483 646.

The message queue functions are:

- “ftok()—Generate IPC Key from File Name” on page 3 (Generate IPC Key from File Name) generates an IPC key based on the combination of path and id.
- “msgctl()—Perform Message Control Operations” on page 8 (Perform Message Control Operations) provides message control operations as specified by cmd on the message queue specified by msqid.
- “msgget()—Get Message Queue” on page 11 (Get Message Queue) returns the message queue identifier associated with the parameter key.
- “msgrcv()—Receive Message Operation” on page 15 (Receive Message Operation) reads a message from the queue associated with the message queue identifier specified by msqid and places it in the user-defined buffer pointed to by msgp.
- “msgsnd()—Send Message Operation” on page 19 (Send Message Operation) is used to send a message to the queue associated with the message queue identifier specified by msqid.
- “QlgFtok()—Generate IPC Key from File Name (using NLS-enabled path name)” on page 23 (Generate IPC Key from File Name (using NLS-enabled path name)) generates an IPC key based on the combination of path and id.

## Semaphore Sets

A **semaphore** is a synchronization mechanism similar to a mutex or a machine interface (MI) lock. It can be used to control access to shared resources, or used to notify other threads of the availability of resources. It differs from a mutex in the following ways:

- A semaphore set is not a single value, but has a set of values. It is referred to through a **semaphore set** containing multiple semaphores. Each semaphore set is identified by a *semid*, which identifies the semaphore set, and a *semnum*, which identifies the semaphore within the set. Multiple semaphore operations may be specified on one **semop()** call. These operations are atomic on multiple semaphores within a semaphore set.
- Semaphore values can range from 0 to 65535.
- Semaphores have **permissions** associated with them. A thread must have appropriate authorities to perform an operation on a semaphore.
- A semaphore can have a **semaphore adjustment value** associated with it. This value represents resource allocations which can be automatically undone by the system when the thread ends, representing the releasing of resources. The adjustment value can range from -32767 to 32767.

Thus, a semaphore can be used as a resource counter or as a lock.

A process gets a semaphore set identifier by calling the **semget()** function. Depending on the *key* and *semflg* parameters passed in, either a new semaphore set is created or an existing semaphore set is accessed. When a new semaphore set is created, a data structure is also created to contain information about the semaphore set. This structure is defined in the `<sys/sem.h>` header file as follows:

```
typedef struct semid_ds {
    struct ipc_perm  sem_perm; /* Permissions structure */
    unsigned short  sem_nsems; /* Number of sems in set */
    time_t          sem_otime; /* Last sem op time */
    time_t          sem_ctime; /* Last change time */
} semtabentry_t;
```

A thread performs operations on one or more of the semaphores in a set by calling the **semop()** function. The following parameters are passed in:

- Semaphore ID
- Pointer to an array of `sembuf` structures
- Number of `sembuf` structures in the array.

The `sembuf` structure is defined in the `<sys/sem.h>` header file as follows:

```
struct sembuf {
    unsigned short sem_num; /* Semaphore number */
    short          sem_op; /* Semaphore operation */
    short          sem_flg; /* Operation flags */
};
```

The operation performed on a semaphore is specified by the `sem_op` field, which can be positive, negative, or zero:

- If `sem_op` is positive, the value of `sem_op` is added to the semaphore's current value.
- If `sem_op` is zero, the caller will wait until the semaphore's value becomes zero.
- If `sem_op` is negative, the caller will wait until the semaphore's value is greater than or equal to the absolute value of `sem_op`. Then the absolute value of `sem_op` is subtracted from the semaphore's current value.

The `sem_flg` value specifies whether or not the thread is willing to wait, and also whether or not the thread wants the system to keep a semaphore adjustment value for the semaphore.

Semaphore waits are visible from the Work with Active Jobs display. A thread waiting on a semaphore in a semaphore set appears to be in a semaphore wait state (SEMW) on the Work with Threads display (requested using the WRKJOB command and taking option 20). Displaying the call stack of the thread shows the **semop()** function near the bottom of the stack.

A thread removes a semaphore set ID by calling the **semctl()** function. The thread also can use the **semctl()** function to change the data structure values associated with the semaphore set ID or to retrieve the data structure values associated with the semaphore set ID. The following parameters are passed in:

- Semaphore set ID
- Command the thread wants to perform (remove ID, set data structure values, receive data structure values),
- Pointer to a buffer from which to set data structure values, or in which to receive data structure values.

In addition, the **semctl()** function can perform various other control operations on a specific semaphore within a set, or on an entire semaphore set:

- Set or retrieve a semaphore value.
- Retrieve the process ID of the last thread to operate on a semaphore.
- Retrieve the number of threads waiting for a semaphore value to increase.
- Retrieve the number of threads waiting for a semaphore value to become zero.
- Retrieve the value of every semaphore in a semaphore set.
- Set the value of every semaphore in a semaphore set.

## Semaphore Set Differences and Restrictions

OS/400 semaphore sets differ from the definition in the Single UNIX Specification in the following ways:

- The Single UNIX Specification does not define threads. Consequently, Single UNIX Specification semaphores are defined in terms of processes and the semaphore:
  - Causes the entire process to wait
  - Releases resources when the process ends

OS/400 handles semaphores at the thread level. An OS/400 semaphore:

- Causes only a single thread to wait
- Releases resources when the thread ends
- The maximum number of semaphore sets that can be created (system-wide) is 2 147 483 646.
- The maximum number of semaphores per semaphore set is 65535.
- Semaphore values are limited to the range from 0 to 65535. Adjustment values associated with a semaphore are limited to the range -32767 to 32767.

The semaphore set functions are:

- “ftok()—Generate IPC Key from File Name” on page 3 (Generate IPC Key from File Name) generates an IPC key based on the combination of path and id.
- “QlgFtok()—Generate IPC Key from File Name (using NLS-enabled path name)” on page 23 (Generate IPC Key from File Name (using NLS-enabled path name)) generates an IPC key based on the combination of path and id.
- “semctl()—Perform Semaphore Control Operations” on page 57 (Perform Semaphore Control Operations) provides semaphore control operations as specified by cmd on the semaphore specified by semnum in the semaphore set specified by semid.
- “semget()—Get Semaphore Set with Key” on page 62 (Get Semaphore Set with Key) returns the semaphore ID associated with the specified semaphore key.
- “semop()—Perform Semaphore Operations on Semaphore Set” on page 65 (Perform Semaphore Operations on Semaphore Set) performs operations on semaphores in a semaphore set. These operations are supplied in a user-defined array of operations.

## Shared Memory

Processes and threads can communicate directly with one another by sharing parts of their memory space and then reading and writing the data stored in the **shared memory**. Synchronization of shared memory is the responsibility of the application program. Semaphores and mutexes provide ways to synchronize shared memory use across processes and threads.

A thread gets a shared memory identifier by calling the **shmget()** function. Depending on the *key* and *shmflg* parameters passed in, either a new shared memory segment is created or an existing shared memory segment is accessed. The size of the shared memory segment is specified by the *size* parameter. When a new shared memory segment is created, a data structure is also created to contain information about the shared memory segment. This structure is defined in the `<sys/shm.h>` header file as follows:

```
typedef struct shmid_ds {
    struct ipc_perm shm_perm; /* Operation permission struct*/
    int shm_segsz; /* Segment size */
    pid_t shm_lpid; /* Process id of last shmop */
    pid_t shm_cpid; /* Process id of creator */
    int shm_nattch; /* Current # attached */
    time_t shm_atime; /* Last shmat time */
    time_t shm_dtime; /* Last shmdt time */
    time_t shm_ctime; /* Last change time */
} shmtabEntry_t;
```

A process gets addressability to the shared memory segment by attaching to it using the **shmat()** function. The following parameters are passed in:

- Shared memory ID
- Pointer to an address
- Flag specifying how the shared memory segment is to be attached

A process detaches a shared memory segment by calling the **shmdt()** function. The only parameter passed in is the shared memory segment address. The process implicitly detaches from the shared memory when the process ends.

A thread removes a shared memory ID by calling the **shmctl()** function. The thread also can use the **shmctl()** function to change the data structure values associated with the shared memory ID or to retrieve the data structure values associated with the shared memory ID. The following parameters are passed in:

- Shared memory ID
- Command the thread wants to perform (remove ID, set data structure values, receive data structure values)
- Pointer to a buffer from which to set data structure values, or in which to receive data structure values.

## Shared Memory Differences and Restrictions

Shared memory segments are created as teraspace-shared memory segments or as nonteraspace-shared memory segments. A teraspace shared memory segment is accessed by adding the shared memory segment to a process's teraspace. A teraspace is a space that has a much larger capacity than other OS/400 spaces and is addressable from only one process. A nonteraspace shared memory segment creates shared memory using OS/400 space objects.

A teraspace shared memory segment is created if SHM\_TS\_NP is specified on the *shmflag* parameter of **shmget()** or if a shared memory segment is created from a program that was compiled using the TERASPACE(\*YES \*TSIFC) option of CRTBNDC or CRTCMOD. The following capabilities and restrictions apply for teraspace shared memory segments.

- Teraspace shared memory objects may be attached in read-only mode.

- The address specified by *shmaddr* is only used when **shmat()** is called from a program that uses data model LLP64 and attaches to a teraspace shared memory segment. Otherwise it is not possible to specify the address in teraspace at which the shared memory is to be mapped. The *shmaddr* parameter on the **shmat()** function is ignored.
- After a teraspace shared memory segment is detached, it cannot be addressed through a pointer saved by the process.
- The maximum size of a teraspace shared memory segment is 4 294 967 295 bytes (4 GB minus 1).
- The maximum number of shared memory segments that can be created (system-wide) is 2 147 483 646.
- A teraspace shared memory segment may be created such that its size can be changed after it is created. The maximum size of this type of shared memory segment is 268 435 456 bytes (256 MB).

The OS/400 nonteraspace shared memory differs from the shared memory definition in the Single UNIX Specification in the following ways:

- The nonteraspace shared memory segments are OS/400 space objects and can be attached only in read/write mode, not in the read-only mode that the Single UNIX Specification allows. If the SHM\_RDONLY flag is specified in the *shmflg* parameter on a **shmget()** call, the call fails and the *errno* variable is set to [EOPNOTSUPP].
- A nonteraspace shared memory segment can be attached only at the actual address of the OS/400 space object, not at an address specified by the thread. The *shmaddr* parameter on the **shmat()** function is ignored.
- After a nonteraspace shared memory segment is detached from a process, it still can be addressed through a pointer saved by the process. For nonteraspace shared memory segments, OS/400 does not "map" and "unmap" regions of storage to the address space of a process.
- The maximum size of a nonteraspace shared memory segment is 16 776 960 bytes. Although the maximum size of a shared memory segment is 16 776 960 bytes, shared memory segments larger than 16 773 120 bytes should be created as teraspace shared memory segments. When the operating system accesses a nonteraspace shared memory segment that has a size larger than 16 773 120 bytes, a performance degradation may be observed.
- The maximum number of shared memory segments that can be created (system-wide) is 2 147 483 646.
- The size of a nonteraspace shared memory segment may be changed using the SHM\_RESIZE command of **shmctl()**, up to a maximum size of 16 773 120 bytes.

The shared memory functions are:

- "ftok()—Generate IPC Key from File Name" on page 3 (Generate IPC Key from File Name) generates an IPC key based on the combination of path and id.
- "QlgFtok()—Generate IPC Key from File Name (using NLS-enabled path name)" on page 23 (Generate IPC Key from File Name (using NLS-enabled path name)) generates an IPC key based on the combination of path and id.
- "shmat()—Attach Shared Memory Segment to Current Process" on page 97 (Attach Shared Memory Segment to Current Process) returns the address of the shared memory segment associated with the specified shared memory identifier.
- "shmctl()—Perform Shared Memory Control Operations" on page 101 (Perform Shared Memory Control Operations) provides shared memory control operations as specified by cmd on the shared memory segment specified by shmid.
- "shmdt()—Detach Shared Memory Segment from Calling Process" on page 104 (Detach Shared Memory Segment from Calling Process) detaches the shared memory segment specified by shmaddr from the calling process.
- "shmget()—Get ID of Shared Memory Segment with Key" on page 106 (Get ID of Shared Memory Segment with Key) returns the shared memory ID associated with the specified shared memory key.

---

## Pointer Based Services

The pointer based services consist of named and unnamed semaphores. The named and unnamed semaphores on OS/400<sup>(R)</sup> differ from the other IPC mechanisms in that they do not have an IPC identifier associated with them. Instead, pointers to the semaphore are used to operate on the semaphore. Before using a semaphore, a process must obtain a pointer to the semaphore. Unlike a semaphore set, a named or unnamed semaphore refers to a single semaphore only. A semaphore contains a value, a maximum value, and a title.

There are two types of semaphores: named semaphores and unnamed semaphores. Once a semaphore is created and a pointer to the semaphore is obtained, the same operations are used to manipulate the values of both types of semaphores. Like the semaphores in a semaphore set, a named or unnamed semaphore has a nonzero value. A semaphore can be used as a resource counter or as a lock. A thread decrements a semaphore to obtain one or more associated resources, and increments the semaphore to release the resource. A semaphore also has a maximum value associated with it. An attempt to increment the value of a semaphore above its maximum value results in an error.

Besides a value, named and unnamed semaphores contain a maximum value and a title. The maximum value sets the highest value that the semaphore value may obtain. The title is a null-terminated string with a maximum size of 16 characters that are associated with the semaphore and may be used to contain debugging information. The titles associated with named and unnamed semaphores may be obtained by using the **QP0ZOLIP()** API.

A process obtains a pointer to a named semaphore by calling the **sem\_open()** or **sem\_open\_np()** functions. These functions find the semaphore associated with a name. The name is a character string, interpreted in the CCSID of the job. The name may be structured so that it looks like a pathname. This name, however, has no relationship to any file system. If the semaphore exists and the process has permission to use the semaphore, then the system allocates memory for the semaphore and returns a pointer to the caller. If the semaphore does not exist, it will be created if the appropriate flags are set. When a new named semaphore is created, the permissions of the semaphore are set using the information provided by the *mode* parameter. These permissions are the same as those used by the identifier-based IPC services. The **sem\_open\_np()** function permits the caller to set the maximum value and title of a semaphore when creating a named semaphore. When a process is finished using a named semaphore, it should call **sem\_close()** to close the semaphore. The semaphore is also explicitly closed when a process terminates. When a named semaphore will no longer be needed, it can be removed from the system using **sem\_unlink()**.

A process obtains a pointer to an unnamed semaphore calling the **sem\_init()** or **sem\_init\_np()** functions. These functions initialize a semaphore at the specified memory location. The **sem\_init\_np()** function permits the caller to set the maximum value and title of a unnamed semaphore when it is created. When a process is finished using an unnamed semaphore, it should call **sem\_destroy()** to destroy the semaphore and release system resources associated with that semaphore.

A process decrements by one the value of a semaphore using the **sem\_wait()** and **sem\_wait\_np()** functions. If the value of the semaphore is currently zero, then the thread is blocked until the value of the semaphore is incremented or until the time specified on the **sem\_wait\_np()** has expired. The **sem\_trywait()** call may be used to decrement the value of the semaphore if it is greater than zero. If the current value of the semaphore is zero, then **sem\_trywait()** will return an error. The **sem\_post()** and **sem\_post\_np()** functions are used to increment the value of a semaphore. After the value of the semaphore is incremented, it may be decremented immediately by threads that have blocked trying to decrement the semaphore.

Named and unnamed semaphore waits are visible from the Work with Active Jobs display. A thread waiting on a named or unnamed semaphore will be in a semaphore wait state (SEMW).

The `sem_getvalue()` function returns the value of the semaphore if the value is greater than or equal to zero. If there are threads waiting on the semaphore, `sem_getvalue()` returns a negative number whose absolute value is the number of threads waiting on the semaphore.

For details on the semaphore functions, see the following:

- “`QlgSem_open()`—Open Named Semaphore (using NLS-enabled path name)” on page 24 (Open Named Semaphore (using NLS-enabled path name)) opens a named semaphore and returns a semaphore pointer that may be used on subsequent calls to `sem_post()`, `sem_post_np()`, `sem_wait()`, `sem_wait_np()`, `sem_trywait()`, `sem_getvalue()`, and `sem_close()`.
- “`QlgSem_open_np()`—Open Named Semaphore with Maximum Value (using NLS-enabled path name)” on page 26 (Open Named Semaphore with Maximum Value (using NLS-enabled path name)) opens a named semaphore and returns a semaphore pointer that may be used on subsequent calls to `sem_post()`, `sem_post_np()`, `sem_wait()`, `sem_wait_np()`, `sem_trywait()`, `sem_getvalue()`, and `sem_close()`.
- “`QlgSem_unlink()`—Unlink Named Semaphore (using NLS-enabled path name)” on page 28 (Unlink Named Semaphore (using NLS-enabled path name)) unlinks a named semaphore.
- “`sem_close()`—Close Named Semaphore” on page 69 (Close Named Semaphore) closes a named semaphore that was previously opened by a thread of the current process using `sem_open()` or `sem_open_np()`.
- “`sem_destroy()`—Destroy Unnamed Semaphore” on page 70 (Destroy Unnamed Semaphore) destroys an unnamed semaphore that was previously initialized using `sem_init()` or `sem_init_np()`.
- “`sem_getvalue()`—Get Semaphore Value” on page 72 (Get Semaphore Value) retrieves the value of a named or unnamed semaphore.
- “`sem_init()`—Initialize Unnamed Semaphore” on page 74 (Initialize Unnamed Semaphore) initializes an unnamed semaphore and sets its initial value.
- “`sem_init_np()`—Initialize Unnamed Semaphore with Maximum Value” on page 75 (Initialize Unnamed Semaphore with Maximum Value) initializes an unnamed semaphore and sets its initial value.
- “`sem_open()`—Open Named Semaphore” on page 78 (Open Named Semaphore) opens a named semaphore, returning a semaphore pointer that may be used on subsequent calls to `sem_post()`, `sem_post_np()`, `sem_wait()`, `sem_wait_np()`, `sem_trywait()`, `sem_getvalue()`, and `sem_close()`.
- “`sem_open_np()`—Open Named Semaphore with Maximum Value” on page 81 (Open Named Semaphore with Maximum Value) opens a named semaphore, returning a semaphore pointer that may be used on subsequent calls to `sem_post()`, `sem_post_np()`, `sem_wait()`, `sem_wait_np()`, `sem_trywait()`, `sem_getvalue()`, and `sem_close()`.
- “`sem_post()`—Post to Semaphore” on page 85 (Post to Semaphore) posts to a semaphore, incrementing its value by one.
- “`sem_post_np()`—Post Value to Semaphore” on page 86 (Post Value to Semaphore) posts to a semaphore, incrementing its value by the increment specified in the options parameter.
- “`sem_trywait()`—Try to Decrement Semaphore” on page 89 (Try to Decrement Semaphore) attempts to decrement the value of the semaphore.
- “`sem_unlink()`—Unlink Named Semaphore” on page 91 (Unlink Named Semaphore) unlinks a named semaphore.
- “`sem_wait()`—Wait for Semaphore” on page 93 (Wait for Semaphore) decrements by one the value of the semaphore.
- “`sem_wait_np()`—Wait for Semaphore with Timeout” on page 95 (Wait for Semaphore with Timeout) attempts to decrement by one the value of the semaphore.

---

## Managing IPC Objects

Interprocess communication objects can be managed with the following APIs. The QP0ZOLIP API opens a list of message queue, shared memory, semaphore set, named semaphore or unnamed semaphore objects by type, by owner, by creator, or by key. The QP0ZOLSM API opens a list of semaphores in a semaphore set. Both APIs return a handle that can be used to get list entries with the QGYGTLE API, find entries by number with the QGYFNDE API, or close the list with the QGYCLST API.

The QP0ZRIPC API retrieves detailed information about message queue, shared memory, or semaphore set objects. The QP0ZDIPC API deletes message queue, shared memory, or semaphore set objects.

The IPC object management APIs are:

- “Delete Interprocess Communication Objects (QP0ZDIPC) API” on page 30 (Delete Interprocess Communication Objects) deletes one or more interprocess communication (IPC) objects as specified by the delete control parameter.
- “Open List of Interprocess Communication Objects (QP0ZOLIP) API” on page 32 (Open List of Interprocess Communication Objects) lets you generate a list of interprocess communication (IPC) objects and descriptive information based on the selection parameters.
- “Open List of Semaphores (QP0ZOLSM) API” on page 44 (Open List of Semaphores) lets you generate a list of description information about the semaphores within a semaphore set.
- “Retrieve an Interprocess Communication Object (QP0ZRIPC) API” on page 47 (Retrieve an Interprocess Communication Object) lets you generate detailed information about a single interprocess communication (IPC) object.

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

---

## Header Files for UNIX-Type Functions

Programs using the UNIX<sup>(R)</sup>-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 [Include files](#) 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
bsdos.h	H	BSEDOS
bseerr.h	H	BSEERR
dirent.h	H	DIRENT
errno.h	H	ERRNO



Name of Header File	Name of File in QSYSINC	Name of Member
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
» qp0lchsg.h	H	QP0LCHSG «
qp0lflop.h	H	QP0LFLOP
qp0ljrnl.h	H	QP0LJRNL
qp0lrer.h	H	QP0LRER
» qp0lrro.h	H	QP0LRRO «
» qp0lrtsg.h	H	QP0LRTSG «
» qp0lscan.h	H	QP0LSCAN «
Qp0lstdi.h	H	QP0LSTDI
qp0wpid.h	H	QP0WPID
qp0zdipc.h	H	QP0ZDIPC
qp0zipc.h	H	QP0ZIPC
qp0zolip.h	H	QP0ZOLIP
qp0zolsm.h	H	QP0ZOLSM
qp0zrirc.h	H	QP0ZRIPC
qp0ztrc.h	H	QP0ZTRC
qp0ztrml.h	H	QP0ZTRML
qp0z1170.h	H	QP0Z1170

Name of Header File	Name of File in QSYSINC	Name of Member
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.

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

---

## Errno Values for UNIX-Type Functions

Programs using the UNIX<sup>(R)</sup>-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.

Name	Value	Text
EBADOPT	3040	Option specified is not valid.
ENOTUPD	3041	File is not opened for update operations.
ENOTDLT	3042	File is not opened for delete operations.
EPAD	3043	The number of characters written is shorter than the expected record length.
EBADKEYLN	3044	A length that was not valid was specified for the key.
EPUTANDGET	3080	A read operation should not immediately follow a write operation.
EGETANDPUT	3081	A write operation should not immediately follow a read operation.
EIOERROR	3101	A nonrecoverable I/O error occurred.
EIORECERR	3102	A recoverable I/O error occurred.
EACCES	3401	Permission denied.
ENOTDIR	3403	Not a directory.
ENOSPC	3404	No space is available.
EXDEV	3405	Improper link.
EAGAIN	3406	Operation would have caused the process to be suspended.
EWOULDBLOCK	3406	Operation would have caused the process to be suspended.
EINTR	3407	Interrupted function call.
EFAULT	3408	The address used for an argument was not correct.
ETIME	3409	Operation timed out.
ENXIO	3415	No such device or address.
EAPAR	3418	Possible APAR condition or hardware failure.
ERECURSE	3419	Recursive attempt rejected.
EADDRINUSE	3420	Address already in use.
EADDRNOTAVAIL	3421	Address is not available.
EAFNOSUPPORT	3422	The type of socket is not supported in this protocol family.
EALREADY	3423	Operation is already in progress.
ECONNABORTED	3424	Connection ended abnormally.
ECONNREFUSED	3425	A remote host refused an attempted connect operation.
ECONNRESET	3426	A connection with a remote socket was reset by that socket.
EDESTADDRREQ	3427	Operation requires destination address.
EHOSTDOWN	3428	A remote host is not available.
EHOSTUNREACH	3429	A route to the remote host is not available.
EINPROGRESS	3430	Operation in progress.
EISCONN	3431	A connection has already been established.
EMSGSIZE	3432	Message size is out of range.
ENETDOWN	3433	The network currently is not available.
ENETRESET	3434	A socket is connected to a host that is no longer available.
ENETUNREACH	3435	Cannot reach the destination network.
ENOBUFS	3436	There is not enough buffer space for the requested operation.
ENOPROTOPT	3437	The protocol does not support the specified option.
ENOTCONN	3438	Requested operation requires a connection.

Name	Value	Text
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.
ERCVERR	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.

Name	Value	Text
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.
EIDINVAL	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.

Name	Value	Text
EJRNDAMAGE	3526	Journal is damaged.
EJRNINACTIVE	3527	Journal is inactive.
EJRNRCVSPC	3528	Journal space or system storage error.
EJRNRMNT	3529	Journal is remote.
ENEWJRNRCV	3530	New journal receiver is needed.
ENEWJRN	3531	New journal is needed.
EJOURNALED	3532	Object already journaled.
EJRNENTTOOLONG	3533	Entry is too large to send.
EDATALINK	3534	Object is a datalink object.
ENOTAVAIL	3535	IASP is not available.
ENOTTY	3536	I/O control operation is not appropriate.
EFBIG2	3540	Attempt to write or truncate file past its sort file size limit.
ETXTBSY	3543	Text file busy.
EASPGRPNOTSET	3544	ASP group not set for thread.
ERESTART	3545	A system call was interrupted and may be restarted.
↳ ESCANFAILURE	3546	An object has been marked as a scan failure due to processing by an exit program associated with the scan-related integrated file system exit points. ⏪

Top | UNIX-Type APIs | APIs by category





---

## Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106-0032, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation  
Software Interoperability Coordinator, Department YBWA  
3605 Highway 52 N  
Rochester, MN 55901  
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

#### COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

---

## Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

Advanced 36  
Advanced Function Printing  
Advanced Peer-to-Peer Networking  
AFP  
AIX  
AS/400  
COBOL/400  
CUA  
DB2  
DB2 Universal Database  
Distributed Relational Database Architecture  
Domino  
DPI

DRDA  
eServer  
GDDM  
IBM  
Integrated Language Environment  
Intelligent Printer Data Stream  
IPDS  
iSeries  
Lotus Notes  
MVS  
Netfinity  
Net.Data  
NetView  
Notes  
OfficeVision  
Operating System/2  
Operating System/400  
OS/2  
OS/400  
PartnerWorld  
PowerPC  
PrintManager  
Print Services Facility  
RISC System/6000  
RPG/400  
RS/6000  
SAA  
SecureWay  
System/36  
System/370  
System/38  
System/390  
VisualAge  
WebSphere  
xSeries

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

---

## Terms and conditions for downloading and printing publications

Permissions for the use of the information you have selected for download are granted subject to the following terms and conditions and your indication of acceptance thereof.

**Personal Use:** You may reproduce this information for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of this information, or any portion thereof, without the express consent of IBM<sup>(R)</sup>.

**Commercial Use:** You may reproduce, distribute and display this information solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of this information, or reproduce, distribute or display this information or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the information or any data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the information is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations. IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THIS INFORMATION. THE INFORMATION IS PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

All material copyrighted by IBM Corporation.

By downloading or printing information from this site, you have indicated your agreement with these terms and conditions.

---

## **Code disclaimer information**

This document contains programming examples.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM<sup>(R)</sup>, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.





Printed in USA