Version 3

Scalable Message Oriented Middleware for Distributed Computing

QueSys / MemSys / SemSys

Reference Manual

Printed in United States of America

X-IPC VERSION 3.3.0

QUESYS/MEMSYS/SEMSYS

REFERENCE MANUAL

TABLE OF CONTENTS

1. UTILITY PROGRAMS

1.1 xipc - X•IPC Interactive Command Processor

NAME

xipc - X•IPC Interactive Command Processor

SYNTAX

xipc

PARAMETERS

None

RETURNS

Not Applicable

DESCRIPTION

xipc is a command interpreter that provides the user with interactive access to X•IPC API capabilities.

Most of the interpreter's commands correspond to X•IPC API's, and their arguments are the same, except for necessary adjustments to the interactive environment. To find a full description of a command and its arguments, refer to the description of the corresponding API.

1.1.1 THE X•IPC INTERACTIVE LANGUAGE

1.1.1.1 SYNTAX

Each command starts with a command verb, usually an X•IPC API name. The command name is followed by the command arguments separated by one or more spaces.

Arguments that consist of a list of values, such as *SidList*, use a comma as a separator between the values.

Text arguments are entered either as a string of characters delimited by spaces or as a string delimited by double quotes. When quotes are used as delimiters, a quote character can also be specified as part of the string by preceding it with a back-slash (\) character.

A line starting with the character "#" is treated as a comment line and its contents are ignored.

1.1.1.2 Variables

xipc provides four sets of built-in variables:

- ACB's - xipc defines 26 ACB variables identified by the letters a through z.
- Message Headers - xipc defines 26 message header variables identified by the letters a through z.

- Memory Sections - `xipc` defines 26 memory section variables identified by the letters `a` through `z`.
- MomSys Message Ids - `xipc` defines 26 message id variables identified by the letters `a` through `z`.

1.1.1.3 Callback Routines

`xipc` has two groups of callback routines that can be used in conjunction with asynchronous operations:

- Six callback routines named `cb1` through `cb6` that display the results of the completing operation.
- Twenty-six callback routines named `cba` through `cbz`. Each of these callback routines can be assigned an `xipc` command to execute when the asynchronous operation completes.

1.1.1.4 Blocking Options

Many of `xipc`'s commands have a blocking option parameter. This parameter corresponds to the blocking option of *X•IPC* API's. The syntax of the blocking option is one of the following:

- **`wait`**
- **`nowait`**
- **`timeout`**(*`Seconds`*)

  `Seconds` - Timeout length in seconds.
- **`callback`**(*`CallbackAction, AcbId`*)

  `CallbackAction` - Either a name of a predefined callback routine (`cb1-cb6` or `cba-cbz`) or an xipc command enclosed in double quotes to be executed when the asynchronous operation completes.

  `AcbId` - ACB variable (a-z).
- **`post`**(*`Sid, AcbId`*)

  `Sid` - Semaphore Id to be set when the operation completes.

  `AcbId` - ACB variable (a-z).
- **`ignore`**(*`AcbId`*)

  `AcbId` - ACB variable (a-z).

Note that all flags must be specified *before* (to the left of) the blocking option.

1.1.1.5 Conventions Used In This Section

The following conventions are used in the description of `xipc` command syntax:

? Text in **bold** is to be entered as specified;

? Items in *italics* represents values to be provided by the user;

? Items between brackets [ ] designate an optional choice.

? Items between braces{} designate a mandatory choice.

1.1.2 GENERAL INTERACTIVE COMMANDS

1.1.2.1 ! - Execute Operating System Command

SYNTAX

! *Command*

ARGUMENTS

Command Native operating system command.

EXAMPLES

```
xipc> # Unix example of operating system command
xipc> !date
Thu May 21 10:58:20 EDT 1996

xipc> # VMS example of operating system command
xipc> !show time
21-May-1996 10:58:20

xipc> # OS/2 example of operating system command
xipc> !date
The current date is: Thu 5-21-1996
Enter the new date: (mm-dd-yy)
```

1.1.2.2 acb - Display Contents of ACB

SYNTAX

acb *AcbId*

ARGUMENTS

AcbId One letter identification of the ACB.

EXAMPLES

```
xipc> acb a
      AUid            = 33
      AsyncStatus     = XIPC_ASYNC_INPROGRESS
      UserData1       = 000000000
      .
      .
      .
```

1.1.2.3 callback - Assign Callback Command

SYNTAX

callback *CallbackName* *XipcCommand*

ARGUMENTS

CallbackName The name of a callback routine (cba-cbz).

XipcCommand An xipc command enclosed in double quotes.

EXAMPLES

```
xipc> # Start spooling when queue fills up
xipc> callback cba "quespool 2 /usr/tmp/spl"
      Command saved
```

1.1.2.4 help - Display List Of Arguments

SYNTAX

help *Command*
? *Command*

ARGUMENTS

Command Name of xipc command.

EXAMPLES

```
xipc> help xipclogin
      xipclogin
         InstanceName
         UserName
```

1.1.2.5 quit - Logout And Quit

SYNTAX
q[uit]

ARGUMENTS
None.

EXAMPLES
```
xipc> q
      Logging out user 11 from: @Server
      Logging out user 31 from: @DBServer
```

1.1.2.6 uid - Display Current User Id

SYNTAX
uid

ARGUMENTS
None.

EXAMPLES
```
xipc> uid
      Uid = 11
```

1.1.3 X^IPC INTERACTIVE COMMANDS

1.1.3.1 xipcabort - Abort a User

SYNTAX

xipcabort *UserId*

ARGUMENTS

UserId User id of user to be aborted

EXAMPLES

```
xipc> xipcabort 11
      RetCode = 0
```

1.1.3.2 xipcconnect - Connect to a Login

SYNTAX

xipcconnect *[InstanceName] [UserId]*

ARGUMENTS

InstanceName Name of instance to connect to: Either an instance configuration file name or an instance name (local or network) starting with the character '@'. Instance name can be specified as '*' in which case the value of the environment variable XIPC will be used. The instance name must be specified exactly as it was specified in the xipclogin command.

UserId User id as returned by xipclogin

EXAMPLES

```
xipc> # Log into stand-alone instance.
      # Disconnect from the login.
      # Then reconnect to the login.
xipc> xipclogin /usr/xipc/test Joe
      Uid = 11
xipc> xipcdisconnect
      RetCode = 0
xipc> xipcconnect /usr/xipc/test 11
      RetCode=0
```

1.1.3.3 xipcdisconnect - Disconnect from a Login

SYNTAX
xipcdisconnect

ARGUMENTS
None

EXAMPLES
```
xipc> xipclogin /usr/xipc/test Joe
      Uid = 11
xipc> xipcdisconnect
      RetCode = 0
xipc> xipclogin /usr/xipc/test2 Joe
      Uid = 7
```

1.1.3.4 xipcerror - Display Error Message

SYNTAX
xipcerror *ErrorCode*

ARGUMENTS
ErrorCode X•IPC error code

EXAMPLES
```
xipc> xipcerror -1003
      Configuration capacity limit exceeded
```

1.1.3.5 xipcfreeze - Freeze Instance

SYNTAX
xipcfreeze

ARGUMENTS
None.

EXAMPLES
```
xipc> xipcfreeze
      RetCode = 0
```

1.1.3.6 xipcgetopt – Get Parameters

SYNTAX

xipcgetopt [*Option*]

ARGUMENTS

[*Option*] One from the following options: CONNECTTIMEOUT, RECVTIMEOUT, PINGTIMEOUT, PINGRETRIES, PINGFUNCTION, PRIVATEQUEUE, MAXTEXTSIZE, ASYNCFD

EXAMPLES

```
xipc> xipcgetopt pingtimeout
      Parameter [pingtimeout] –> : [5]
```

1.1.3.7 xipcidlewatch - Control Idle Watch Monitoring

SYNTAX

Xipcidlewatch *[Option]*

ARGUMENTS

Option One of “start,” “stop” or “mark.”

EXAMPLES

```
xipc> xipcidlewatch start
      RetCode = 0
```

1.1.3.8 xipcinfologin - Get Login Information

SYNTAX
xipcinfologin

ARGUMENTS
None

EXAMPLES
```
xipc> xipcinfologin
      Uid    Instance
      ---    --------
      11    /usr/xipc/test
       7    @Server
      31    @DBServer
```

1.1.3.9 xipcinfoversion - Get XIPC Version Information

SYNTAX
xipcinfoversion | xipcver

ARGUMENTS
None

EXAMPLES
```
xipc> xipcinfoversion
      XIPC Version 3.1.0 (GA) ba - Windows NT 4.0
```

1.1.3.10 xipcinit – Initiate Platform Environment

SYNTAX
xipcinit

ARGUMENTS
None

EXAMPLES
```
xipc> xipcinit
      xipcinit: XIPC Platform Environment Initiated
      Win32 – XIPC 3.3.0aa [Build 5000]
      RetCode = 0
```

1.1.3.11 xipclist - List Active Network Instances

SYNTAX

xipclist *[NodeName]*

ARGUMENTS

NodeName Name of node about which xipclist's reporting should be limited.

EXAMPLES

```
xipc> xipclist
      Machine.................[grumpy]
      Instance Name...........[server]
      Instance File Name......[/xipc/server]
      Maximum Text Size.......[1024]
```

1.1.3.12 xipclogin - Log Into An Instance

SYNTAX

xipclogin *[InstanceName] [UserName]*

ARGUMENTS

InstanceName Name of instance to log into: Either an Instance File Name or an instance name (local or network) starting with the character '@'. Instance name can be specified as '*' in which case the value of the environment variable XIPC will be used.

UserName Name to be assigned to the *X•IPC* user. If omitted, the string XIPC will be used. If the name "superuser" is used, the user is logged in as a superuser.

EXAMPLES

```
xipc> xipclogin  /tmp/config George
      Uid = 11

xipc> # Log into instance "Server". "xipc" is default user name
xipc> xipclogin  @Server
      Uid = 1

xipc> # Log into network instance on node "dopey"
xipc> xipclogin  @dopey:Server George
      Uid = 1
```

1.1.3.13 xipclogout - Log Out Of Instance

SYNTAX
`xipclogout`

ARGUMENTS
None.

EXAMPLES
```
xipc> xipclogout
      RetCode = 0
```

1.1.3.14 xipcmasktraps - Activate Trap Mask

SYNTAX
`xipcmasktraps`

ARGUMENTS
None.

EXAMPLES
```
xipc> xipcmasktraps
      RetCode = 0
```

1.1.3.15 xipcsetopt - Set Parameters

SYNTAX
`xipcsetopt [Option]`

ARGUMENTS

[*Option*]	One from the following options: CONNECTTIMEOUT, RECVTIMEOUT, PINGTIMEOUT, PINGRETRIES, PINGFUNCTION, PRIVATEQUEUE, MAXTEXTSIZE, ASYNCFD

EXAMPLES
```
xipc> xipcsetopt pingtimeout 5
      Parameter [pingtimeout] –> New value [5]
```

1.1.3.16 xipcstart - Start An Instance

SYNTAX

xipcstart *InstFileName InstName [Options]*

ARGUMENTS

InstFileName The instance configuration file name of instance to be started (i.e., the path name of its instance configuration file). The Instance File Name can be omitted, in which case the value of the environment variable `XIPC` will be used.

InstName Name to be assigned to the instance. The Instance Name can be omitted, in which case the optional value (`LOCALNAME` or `NETNAME`) in the [`XIPC`] section of the Instance Configuration File may be used. Note that, should the Instance Name be omitted, and if `local` is not specified, `network` is the default (as shown in the example below). (See *`[Options]`* below for setting an instance as `local` or `network`.)

Note that *X•IPC* instances that are started with an assigned name (either a *Local* or a *Network* name) are visible to the `xipclist` utility command. It is sometimes desirable that an instance's existence not be visible to `xipclist`. This can be accomplished by assigning the instance a name starting with the '_' (underscore) character. So for example: an instance named `foo` would be visible to `xipclist`, while an instance named `_foo` would not.

[Options] One or more of the following: `initialize, network, local, report, test` or `0`. When listing multiple options, they are listed and separated by commas.

Note: Asterisks (*) can be used as "place holders," with the defaults noted above, if the arguments preceding *`[Options]`* are not specified. See the example below.

EXAMPLES

```
xipc> # Start a Network instance
xipc> xipcstart  /tmp/config  Server
      XipcStart(SemSys):
      .
      .
      XipcReg: Network Instance [Server] Registered.
      RetCode = 0

xipc> # Start a Stand-Alone Instance.
xipc> # Use XIPC environment variable to specify instance name.
xipc> # Do not output report.
xipc> xipcstart  * * 0
      RetCode = 0
```

1.1.3.17 xipcstop - Stop An Instance

SYNTAX

`xipcstop` *`InstanceName [Options]`*

ARGUMENTS

`InstanceName` Name of instance to be stopped: Either an Instance File Name or an instance name (local or network) starting with the character '@'. The instance name can be omitted and specified as '*', in which case the value of the environment variable `XIPC` will be used.

`Options` One of: `report, force` or `0`.

EXAMPLES

```
xipc> # Use XIPC environment variable to specify instance name.
xipc> # Do not output report.
xipc> xipcstop  * 0
      RetCode = 0
```

1.1.3.18 xipcterm – Terminate Platform Environment

SYNTAX

`xipcterm`

ARGUMENTS

None

EXAMPLES

```
xipc> xipcterm
      xipcterm:XIPC Platform Environment Terminated
      RetCode = 0
```

1.1.3.19 xipcunfreeze - Unfreeze Instance

SYNTAX
`xipcunfreeze`

ARGUMENTS
None.

EXAMPLES
```
xipc> xipcunfreeze
      RetCode = 0
```

1.1.3.20 xipcunmasktraps - Deactivate Trap Mask

SYNTAX
`xipcunmasktraps`

ARGUMENTS
None.

EXAMPLES
```
xipc> xipcunmasktraps
      RetCode = 0
```

1.2 queview - View an Instance's QueSys

NAME

queview - View an Instance's QueSys

SYNTAX

queview [*Interval*] [*InstName*]

PARAMETERS

Name	Description
Interval	The initial time interval between screen updates (in milliseconds). The default value is 1000.
InstName	The instance file name of the instance or the registered name of the instance to be monitored.

RETURNS

Value	Description

No return value.

DESCRIPTION

This program is used for real-time monitoring of the activities occurring within an instance's QueSys. The specified *InstName* identifies the instance to be monitored. If *InstName* is not specified, the value of the "XIPC" environment variable is used as the instance file name of the instance to be monitored. While queview is running, it is possible to control its operation by entering commands. The "command key" is operating system dependent and is defined in the respective Platform Notes. (As an example, on most Unix platforms, the terminal "interrupt" key is the "command" key.)
At the Command> prompt, one of the following commands can be entered. (Text in **bold** is to be typed in as specified; items in *italics* represent values to be provided by the user.)

i*n*	Set time interval to *n* milliseconds. Example: i100.
zu*n*	Zoom in on user *n*. Example: zu15.
zq*n*	Zoom in on queue *n*. Example: zq7.
zm*n*	Zoom in on the messages on queue *n*, display by time strand. Example: zm8.
zm*n***t**	Zoom in on the messages occurring on queue *n*, display by time strand. Example: zm8t.
zm*n***p**	Zoom in on the messages occurring on queue *n*, display by priority strand. Example: zm4p.
zs*n*	Zoom in on the spool activity of queue *n*. Example: zs9.
zp	Zoom in on message text pool. Example: zp.
uz	Un-zoom, close the zoom window.
pu*n*	Pan view to user *n*. Example: pu3
pq*n*	Pan view to queue *n*. Example: pq10
tf	Enter Trace Flow mode.

ts	Enter Trace Step mode.
to	Turn Trace off.
b*n*	Browse the messages on queue *n*, following the time strand. See below for browse commands. Example: `b8.`
b*n***t**	Browse the messages on queue *n*, following the time strand. See below for browse commands. Example: `b8t.`
b*n***p**	Browse the messages on queue *n*, following the priority strand. See below for browse commands. Example: `b4p`.
q	Quit.

Browse facility commands:

p	Switch display to follow the priority strand.
t	Switch display to follow the time strand.
f	Move to the first message on the current strand.
l	Move to the last message on the current strand.
b*n*	Browse the messages on queue *n*, following the time strand. Example: `b8.`
b*n***t**	Browse the messages on queue *n*, following the time strand. Example: `b8t.`
b*n***p**	Browse the messages on queue *n*, following the priority strand. Example: `b4p`.
q	Quit.

Navigating on a Queue:

⇒ (right arrow)	Move to the next message on the current strand.
⇐ (left arrow)	Move to the previous message on the current strand.
n	Move to the nth message on the current strand.
+n	Move forward n messages.
-n	Move backward n messages.

Navigating within a message:

⇑ (up arrow)	Scroll the current message up one line.
⇓ (down arrow)	Scrolls the current message down one line.
PAGE-UP	Scroll the current message one page up.
PAGE-DOWN	Scroll the current message one page down.
HOME	Scroll the current message to its top.
END	Scroll the current message to its bottom.

ASCII pattern searching:

/IBM/	Search forward in the current message for the ASCII string "IBM".
//	Repeat the search.
/	Same.
\IBM\	Search backwards in the current message for the ASCII string "IBM".
\\	Repeat the search.
\	Same.
g/IBM/	Search forward for "IBM" through all messages to the end of the queue.
g//	Repeat the search.
g/	Same.
g\IBM\	Search backwards for "IBM" through all messages to the start of the queue.
g\\	Repeat the search.
g\	Same.

Hexadecimal pattern searching:

`/4f37/x`	Search forward for the hex pattern "4f37" within the current message.
`g/4f37/x`	Same search, but forward through all messages on the queue.
`g//x`	Same.
`\4f37\x`	Searches backwards for the hex pattern "4f37" within the current message.
`g\4f37\x`	Same search, but backwards through all messages on the queue.
`g\\x`	Same.

ERRORS
Display messages.

1.3 memview - View an Instance's MemSys

NAME
memview - View an Instance's MemSys

SYNTAX
memview [*Interval*] [*InstName*]

PARAMETERS

Name	Description
Interval	The initial time interval between screen updates (in milliseconds). The default value is 1000.
InstName	The instance file name of the instance or the registered name of the instance to be monitored.

RETURNS

Value	Description

No return value.

DESCRIPTION
This program is used for real-time monitoring of the activities occurring within an instance of MemSys. The specified *InstName* identifies the instance to be monitored. If *InstName* is not specified, the value of the "XIPC" environment variable is used as the instance file name of the instance to be monitored. While memview is running, it is possible to control its operation by entering commands. The "command key" is operating system dependent and is defined in the respective Platform Notes. (As an example, on most Unix platforms, the terminal "interrupt" key is the "command" key.)
At the Command> prompt, one of the following commands can be entered. (Text in **bold** is to be typed in as specified; items in italics represent values to be provided by the user.)

i*n*	Set interval to *n* milliseconds. Example: i100.
zu*n*	Zoom in on user *n*. Example: zu15.
zm*n*	Zoom in on memory segment *n*. Example: zm7.
zp	Zoom in on memory text pool. Example: zp.
uz	Un-zoom, close the zoom window.
s*n*	Monitor section activity on memory segment *n*. Example: s0
w*n*	Watch memory segment *n*. Example: w2
pu*n*	Pan view to user *n*. Example: pu3
pm*n*	Pan view to memory segment *n*. Example: pm10
tf	Enter Trace Flow mode.
ts	Enter Trace Step mode.
to	Turn Trace off.

b*n* Browse memory segment *n*. Example: `b8`.

q Quit.

Section window commands:

i*n* Set time interval to *n* milliseconds. Example: `i100`.

s*n* Monitor section activity on memory segment *n*. Example: `s0.`

w*n* Watch memory segment *n*. Example: `w2`

tf Enter Trace Flow mode.

ts Enter Trace Step mode.

to Turn Trace off.

b*n* Browse memory segment *n*. Example: `b8`.

q Quit.

Navigating within the section monitor window:

⇑ (up arrow)	Scroll up one line.
⇓ (down arrow)	Scroll down one line.
PAGE-UP	Scroll up one page.
PAGE-DOWN	Scroll down one page.
HOME	Scroll to the top of the section data.
END	Scroll to the bottom of the section data.

Watch window commands:

i*n* Set time interval to *n* milliseconds. Example: `i100.`

s*n* Monitor section activity on memory segment *n*. Example: `s0`

w*n* Watch memory segment *n*. Example: `w2`

tf Enter Trace Flow mode.

ts Enter Trace Step mode.

to Turn Trace off.

b*n* Browse memory segment *n*. Example: `b8.`

q Quit.

Navigating within the segment watch window:

⇑ (up arrow)	Scroll up one line (20 Bytes).
⇓ (down arrow)	Scroll down one line (20 Bytes).
PAGE-UP	Scroll up one page (260 Bytes).
PAGE-DOWN	Scroll down one page (260 Bytes).
HOME	Scroll to the top of the memory segment.
END	Scroll to the bottom of the memory segment.

Browse facility commands:

b*n* Browse memory segment *n*. Example: `b8.`

q Quit.

Navigating within a memory segment:

⇑ (up arrow) Scrolls up one line (20 Bytes).

⇓(down arrow)	Scrolls down one line (20 Bytes).
PAGE-UP	Scrolls up one page (260 Bytes).
PAGE-DOWN	Scrolls down one page (260 Bytes).
HOME	Scrolls to the top of the memory segment.
END	Scrolls to bottom of the memory segment.

ASCII pattern searching:

/IBM/	Search forward for the ASCII string "IBM".
//	Repeat the search.
///	Same.
\IBM\	Search backwards for the ASCII string "IBM".
\\	Repeat the search.
\\\	Same.

Hexadecimal pattern searching:

/4f37/x	Search forward for the hex pattern "4f37".
\4f37\x	Searches backwards for the hex pattern "4f37".

ERRORS
Display messages.

1.4 semview - View an Instance's SemSys

NAME
semview - View an Instance's SemSys

SYNTAX
```
semview [Interval] [InstName]
```

PARAMETERS

Name	Description
Interval	The initial time interval between screen updates (in milliseconds). The default value is 1000.
InstName	The instance file name of the instance or the registered name of the instance to be monitored.

RETURNS

Value	Description
No return value.	

DESCRIPTION
This program is used for real-time monitoring of the activities occurring within an instance's SemSys. The specified *InstName* identifies the instance to be monitored. If *InstName* is not specified, the value of the "XIPC" environment variable is used as the instance file name of the instance to be monitored. While `semview` is running, it is possible to control its operation by entering commands. The "command key" is operating system dependent and is defined in the respective Platform Notes. (As an example, on most Unix platforms, the terminal "interrupt" key is the "command" key.)
At the `Command>` prompt, one of the following commands can be entered. (Text in **bold** is to be typed in as specified; items in *italics* represent values to be provided by the user.)

i*n*	Set time interval to *n* milliseconds. Example: `i100.`
zu*n*	Zoom in on user *n*. Example: `zu15.`
zs*n*	Zoom in on semaphore *n*. Example: `zs7`.
uz	Un-zoom, close the zoom window.
pu*n*	Pan view to user *n*. Example: `pu3`
ps*n*	Pan view to semaphore *n*. Example: ps10
tf	Enter Trace Flow mode.
ts	Enter Trace Step mode.
to	Turn Trace off.
q	Quit.

ERRORS
Display messages.

2. QUESYS PARAMETERS, FUNCTIONS AND MACROS

2.1 X·IPC Instance Configuration - QueSys Parameters

NAME

X·IPC **Instance Configuration** - QueSys parameter definitions for .cfg files

SYNTAX

```
[QUESYS]
General QueSys parameters, defined below
```

PARAMETERS

The table below lists the general QueSys configuration parameters, including the parameter name, description and default value. The order in which parameters appear within the `[QUESYS]` section of the configuration is not significant. The default values shown do *not* represent limits for the values that any particular user may require.

Parameter Name	Description	Default Value
MAX_QUEUES	The number of concurrent queues. It should be set based on the requirements of the programs using the instance.	16
MAX_USERS	The maximum number of concurrent QueSys users (real users and pending asynchronous operations) that can be supported by the subsystem. It should be set based on the requirements of the programs using the instance. Note that asynchronously blocked QueSys operations are treated as QueSys users. The expected level of QueSys asynchronous activity should therefore be factored into this parameter.	32
MAX_NODES	The maximum number of nodes. QueSys nodes are used internally for tracking users that block on QueSys operations. The value depends largely on the nature of the program that will use the instance. A conservative estimate can be calculated with the following formula: **MAX_NODES = MAX_QUEUES + (MAX_USERS*3) + (MAX_USERS*MAX_QUEUES)** The default value was calculated using the **MAX_QUEUES** and **MAX_USERS** defaults.	624

Parameter Name	Description	Default Value
MAX_HEADERS	The maximum number of concurrent message headers (i.e., messages) that can be circulating within an instance at any one time. A conservative estimate can be calculated with the following formula: **MAX_HEADERS = MAX_QUEUES + (MAX_QUEUES *** *AverageQueueLength*) where: *AverageQueueLength* is the expected average queue length (in terms of messages) within the instance. The default value was calculated using the above default values and an *AverageQueueLength* of *5*.	96
SIZE_MSGPOOL	The size of the message text pool (K-Bytes). QueSys provides optional blocking when accessing the message pool. Consequently, a less conservative approach can be applied when configuring the message text pool. A starting value can be calculated with the following formula: **SIZE_MSGPOOL = (MAX_QUEUES *** *AverageQueueLength*) * (*AverageMessageSize*+16) where: *AverageQueueLength* is as defined above, and *AverageMessageSize* is the expected average message size occurring within the instance. The default value was calculated using the default values above, an *AverageQueueLength* of *5* and an *AverageMessageSize* of *256*. **SIZE_MSGPOOL** is expressed in terms of K-bytes. As such the calculated value should be rounded up to the next K-byte multiple. (E.g., if the calculation equals 1948 bytes, then 2K bytes should be specified.)	22
SIZE_MSGTICK	The message text pool allocation size unit. This value specifies the multiple by which all text pool allocations are made. A proper value can have a noticeable effect in reducing fragmentation in the message pool. **SIZE_MSGTICK** should be rounded up to a multiple of 4. A starting value can be calculated with the following formula: **SIZE_MSGTICK**=*25PercentileMessageSize* where: *25PercentileMessageSize* is defined as the size value for which it is expected that 75% of the instance's messages will be larger in size and 25% will be smaller.	32

Parameter Name	Description	Default Value
SIZE_SPLTICK	The spool tick file size limit (K-bytes). It defines the file size limit used in queue overflow spooling. The QueSys spooling mechanism uses one or more files to handle each queue's message spooling. **SIZE_SPLTICK** sets the maximum size of these files (in K-bytes). Too large a value could result in wasted file system space, holding a queue's old spooled data; too small a value will generally cause a greater number of spool files to be created for each queue. The selection of a value depends on which of the competing concerns is more important. If the value for **SIZE_SPLTICK** is being chosen to meet a system-wide file size limit, then a smaller value (less than the system file size limit) should be chosen. If the concern is to limit spool file proliferation, then a large value will be appropriate. In either case, **SIZE_SPLTICK** must be at least 32 bytes larger than the largest message to be spooled by any queue in the instance.	32

2.2 Functions

2.2.1 QueAbortAsync() - ABORT AN ASYNCHRONOUS OPERATION

NAME

`QueAbortAsync()` - Abort An Asynchronous Operation

SYNTAX

```
#include "xipc.h"


XINT
QueAbortAsync(AUid)

XINT AUid;
```

PARAMETERS

Name	Description
AUid	The asynchronous operation User ID of the operation to be aborted.

RETURNS

Value	Description
RC >= 0	Abort successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueAbortAsync() aborts a pending asynchronous operation.
If the aborted asynchronous operation was issued by the same X•IPC user, the *BlockOpt* of the aborted operation is ignored and the Asynchronous Result Control Block is not set.
If the aborted operation was issued by a different user, a return code of `QUE_ER_ASYNCABORT` is placed in the *RetCode* field of the operation's Asynchronous Result Control Block and the action specified in the *BlockOpt* of the aborted operation is carried out, i.e., a callback routine is invoked or a semaphore is set.

ERRORS

Code	Description
`QUE_ER_BADUID`	Invalid *AUid* parameter.
`QUE_ER_NOSUBSYSTEM`	QueSys is not configured in the instance.
`QUE_ER_NOTLOGGEDIN`	User not logged into instance (User never logged in, was aborted or disconnected).
`QUE_ER_SYSERR`	An internal error has occurred while processing the request.

XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

queabortasync *AsyncUserId*

ARGUMENTS

AsyncUserId Asynchronous user id of the asynchronous QueSys operation to be aborted

EXAMPLES

```
xipc> quereceive hp 0 callback(cb1,q)
      RetCode = -1097
      Operation continuing asynchronously
xipc> acb q
      AUid = 35
      .
      .
xipc> queabortasync 35
......Callback function CB1 executing......
      RetCode = -1098
      Asynchronous operation aborted
      .
      .
      .
```

2.2.2 QueAccess() - ACCESS AN EXISTING QUEUE

NAME

QueAccess() - Access an Existing Queue

SYNTAX

```
#include "xipc.h"


XINT
QueAccess(Name)

CHAR *Name;
```

PARAMETERS

Name	Description
Name	A pointer to a string that contains the symbolic name identifying the desired queue. The *Name* must be null terminated, must not exceed QUE_LEN_XIPCNAME characters, must identify an existing queue and cannot be QUE_PRIVATE.

RETURNS

Value	Description
RC >= 0	Access successful. RC is Queue ID (Qid). Qid is to be used in all subsequent QueSys calls that refer to this queue.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueAccess() accesses an existing queue in QueSys. *Name* is used for identifying the desired queue. The function returns Qid of the accessed queue.

ERRORS

Code	Description
QUE_ER_BADQUENAME	Invalid *Name* parameter.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTFOUND	Queue with *Name* does not exist.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).

XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

queaccess *QueName*

ARGUMENTS

QueName Name of queue

EXAMPLES

```
xipc> queaccess ServerQ
      Qid = 7
```

2.2.3 QueBrowse() - BROWSE A MESSAGE QUEUE

NAME

QueBrowse() - Browse a Message Queue

SYNTAX

```
#include "xipc.h"


XINT
QueBrowse(MsgHdr, Direction)

MSGHDR *MsgHdr;
XINT Direction;
```

PARAMETERS

Name	Description
MsgHdr	A pointer to a message header variable that contains a copy of a message header still residing on a queue.
Direction	The direction of the browse operation.

RETURNS

Value	Description
RC >= 0	Browse successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueBrowse() returns with a copy of the message header one position in the specified *Direction*, relative to the message header identified by the *MsgHdr* parameter. *MsgHdr* must reference a message header that has not been dequeued. *MsgHdr* may have been accessed through a call to QueGet(), specifying the QUE_NOREMOVE option (to the left of the blocking option) or through a previous call to QueBrowse().

Possible values for the direction parameter are:

QUE_PRIO_NEXT	Access the next header on the priority strand (decreasing priority).
QUE_PRIO_PREV	Access the previous header on the priority strand (increasing priority).
QUE_TIME_NEXT	Access the next header on the time strand (more recent).
QUE_TIME_PREV	Access the previous header on the time strand (less recent).

QueBrowse() will fail, returning QUE_ER_ENDOFQUEUE if no additional message headers exist in the specified direction.

ERRORS

Code	Description
QUE_ER_BADDIRECTION	Invalid *Direction* parameter.
QUE_ER_BADTEXT	*MsgHdr* has invalid text pointer.
QUE_ER_ENDOFQUEUE	An end of the queue has been reached.
QUE_ER_MSGHDRREMOVED	*MsgHdr* has been removed from queue.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

quebrowse *MsgHdr Direction*

ARGUMENTS

MsgHdr A one letter message header variable.

Direction One of: **time+**, **time-**, **prio+**, **prio-**.

EXAMPLES

```
xipc> queget a hp 0 noremove,wait
      RetCode = 0, Qid = 0, Seq# = 1221, Prio = 100, HdrStatus = NOT-REMOVED
xipc> quebrowse a time+
      RetCode = 0, Qid = 0, Seq# = 1233, Prio = 100, HdrStatus = NOT-REMOVED
xipc> quebrowse a time+
      RetCode = -1625
      An end of the queue has been reached
```

2.2.4 QueBurstSend() - SEND A BURST MESSAGE TO A QUEUE

NAME

QueBurstSend() - Send a Burst Message to a Queue

SYNTAX

```
#include "xipc.h"


XINT
QueBurstSend(TargetQid, MsgBuf, MsgLength, Priority)

XINT TargetQid;
XANY *MsgBuf;
XINT MsgLength;
XINT Priority;
```

PARAMETERS

Name	Description
TargetQid	Identifies a target Qid for this QueBurstSend() operation. *TargetQid* overrides any queue targeting specified at the start of the send-burst, via QueBurstSendStart(). Specifying QUE_NULL_QID directs X•IPC to use the original queue targeting.
MsgBuf	A pointer to the message text to be sent.
MsgLength	The size (in bytes) of the message in *MsgBuf*. *MsgLength* must be greater than 0.
Priority	An integer to be designated as the message's priority. *Priority* must be greater than 0.

RETURNS

Value	Description
RC > 0	Sequence number of message within current send-burst.
RC < 0	Error (see error codes below).

DESCRIPTION

QueBurstSend() sends a message onto a queue as part of a send-burst. The send-burst must have been previously started using the QueBurstSendStart() function call.
QueBurstSend() selects a target queue based on the *QueSelectCode* and *QidList* parameters specified at the start of the send-burst (i.e., within the QueBurstSendStart() function call). It is possible to override that targeting mechanism by specifying a valid *TargetQid* within the QueBurstSend() call. Otherwise, *TargetQid* should be specified as QUE_NULL_QID.
QueBurstSend() will attempt to enqueue the message pointed at by *MsgBuf*. *MsgLength* must not exceed the *MaxMsgLength* value specified in the burst's initiating call to QueBurstSendStart(). Should the enqueuing operation need to block (e.g., due to a queue capacity limitation), then QueBurstSend() will perform the *BlockingOption* as specified at the start of the send-burst (i.e., within the

QueBurstSendStart() function call). The QueBurstSend() operation returns a burst sequence number that uniquely identifies the sent message within the current send-burst.
Error reporting within a send-burst can occur in one of two ways, depending on the error:

- Synchronous errors are those that are detected within the call to QueBurstSend(), and are reported on immediately within the return code value returned by QueBurstSend(). These errors are identified below with an [S] notation to indicate their synchronous nature.
- Messages passed to QueBurstSend() may not be immediately enqueued. This is especially the case when a network is involved. Because of this latency between the delivery step and the enqueuing step, errors that occur during the enqueuing step are reported asynchronously using the asynchronous *ErrorOption* specified within the QueBurstSendStart() call at the start of the burst. Error information is returned within the ACB specified as part of the *ErrorOption*.

  Information within the error reporting ACB are the burst message sequence number of the message that failed to be enqueued and an *X♦IPC* error code describing the enqueuing error.

  Errors that are reported asynchronously are identified below with an [A] notation to indicate their asynchronous nature. Note that certain errors may be reported in either way, depending on the specific nature of the error.

If an error occurs, the send-burst is terminated. It is an error to issue subsequent QueBurstSend() operations without starting a new send-burst.
For a more detailed discussion of QueBurstSend(), refer to the presentation in the Advanced Topics chapter of the QueSys/MemSys/SemSys User Guide.

ERRORS

Note: `[S]` preceding the error code description indicates an error that is reported synchronously; `[A]` preceding the error code description indicates an error that is reported asynchronously. Certain errors, with both `[A]` and `[S]`, may be reported in either way, depending on the specified nature of the error.

Code	Description
`QUE_ER_BADBUFFER`	`[S]` *MsgBuf* is `NULL`.
`QUE_ER_BADLENGTH`	`[S]` Invalid *MsgLength* parameter.
`QUE_ER_BADQID`	`[A]` Bad *TargetQid*, or `QUE_NULL_QID` was specified when valid Qid is required.
`QUE_ER_CAPACITY_ASYNC_USER`	`[S]` QueSys async user table full
`QUE_ER_CAPACITY_HEADER`	`[S]` QueSys header table full
`QUE_ER_CAPACITY_NODE`	`[S]` QueSys node table full
`QUE_ER_DESTROYED`	`[A]` Another user destroyed a queue that a blocked QueBurstSend() call was waiting to enqueue onto. The blocked QueBurstSend() operation was canceled. No message was enqueued.
`QUE_ER_INTERRUPT`	`[S]` A QueBurstSend() operation that had blocked due to the underlying protocol's flow-control, was interrupted by a signal. The message was not sent.
`QUE_ER_ISFROZEN`	`[A]` User froze instance or QueSys since starting send-burst. A QueBurstSend() operation was about to block based on the burst's originally specified *BlockingOption* of `QUE_WAIT` or QUE_TIMEOUT.

QUE_ER_NOSUBSYSTEM	[A] QueSys is not configured in the instance.
QUE_ER_NOTINSENDBURST	[S] User not in send-burst.
QUE_ER_NOTLOGGEDIN	[S] [A] User not logged into instance (User never logged in, was aborted or disconnected).
QUE_ER_NOWAIT	[A] The originally specified *BlockingOption* for this send-burst was QUE_NOWAIT. The enqueuing step of a QueBurstSend() operation could not be immediately satisfied (e.g., queue was full).
QUE_ER_PURGED	[A] Another user purged a queue that the blocked QueBurstSend() call was waiting on. The blocked QueBurstSend() operation was canceled. No message was enqueued.
QUE_ER_TIMEOUT	[A] The originally specified *BlockingOption* for this send-burst was QUE_TIMEOUT. The enqueuing step of a QueBurstSend() operation could not be satisfied during the timeout period (e.g., queue was full).
QUE_ER_TOOBIG	[S] The size of the message exceeds the *MaxMsgLength* in the burst's originating call to QueBurstSendStart().
XIPCNET_ER_CONNECTLOST	[S] Connection to instance lost.
XIPCNET_ER_NETERR	[S] Network transmission error.
XIPCNET_ER_SYSERR	[S] Operating system error.

INTERACTIVE COMMAND

SYNTAX

{queburstsend | qbs} *TargetQid Message Priority*

ARGUMENTS

TargetQid Identifies a target Qid for this QueBurstSend operation. *TargetQid* overrides any queue targeting specified at the start of the send-burst, via QueBurstSendStart. Specifying **null** directs *X♦IPC* to use the original queue targeting.

MsgBuf Message text to be sent.

Priority An integer to be designated as the message's priority. *Priority* must be greater than 0.

EXAMPLES

```
xipc> queburstsend 3 hello 123
      SeqNo = 556
xipc> qbs 1 “hello again” 456
      SeqNo = 557
```

2.2.5 QueBurstSendStart() - START A SEND-BURST

NAME

QueBurstSendStart() - Start a Send-Burst

SYNTAX

```
#include "xipc.h"


XINT
QueBurstSendStart(QueSelectCode, QidList, MaxMsgLength,
                  ReadAheadBufSize, BlockingOption, ErrorOption)

XINT QueSelectCode;
QIDLIST QidList;
XINT MaxMsgLength;
XINT ReadAheadBufSize;
... BlockingOption;
... ErrorOption;
```

PARAMETERS

Name	Description
QueSelectCode	A code indicating the queue selection criteria to be used for targeting QueBurstSend() operations during the upcoming send-burst. The selected queue is one of the Qids in *QidList*. Possible values for *QueSelectCode* are provided in the companion User Guide. It is also possible to specify QUE_NULL_QUESELECTCODE. Doing soindicates that all QueBurstSend() calls within the upcoming send-burst will specify a non-null *TargetQid*. (Refer to the QueBurstSend() function call for details.)
QidList	A list of Qids for consideration as target queue of QueBurstSend() operations within the upcoming send-burst. A QIDLIST is constructed using QueList() or QueListBuild() and is updated using QueListAdd(). A pointer to a QIDLIST (type PQIDLIST) may be passed as well. It is additionally possible to specify QUE_NULL_QIDLIST. Doing so indicates that all QueBurstSend() calls within the upcoming send-burst will specify a non-null *TargetQid*. (Refer to the QueBurstSend() function call for details.)
MaxMsgLength	The maximum size (in bytes) of messages to be sent in the upcoming send-burst.
ReadAheadBufSize	The size in bytes of the read-ahead buffer used by *X•IPC* to read-ahead messages off the network. This value may be set to QUE_BURST_DEFAULT_READAHEADSIZE, in which case *X•IPC* uses determines a value based on underlying protocol configuration settings.

BlockingOption The blocking option to be used when enqueuing messages during QueBurstSend() operations of the upcoming send-burst. Valid options are: QUE_WAIT, QUE_NOWAIT and QUE_TIMEOUT. Asynchronous blocking options are not allowed. Refer to the Advanced Topics section of the companion Reference Manual for a description of these options.

ErrorOption The QUE_CALLBACK() option specifying how enqueue error conditions should be reported during the upcoming send-burst. Error information is reported asynchronously within the specified option's associated ACB structure.

RETURNS

Value	Description
RC >= 0	Burst initialization and start was successful.
RC < 0	Error (see error codes below).

DESCRIPTION

QueBurstSendStart() starts a send-burst. QueBurstSendStart() must be called before any burst messages can be sent. Once a send-burst is started, messages are sent via the QueBurstSend() operation. The send-burst is eventually terminated by a call to the QueBurstSendStop() function, or when an error condition is encountered.

The *QueSelectCode* and *QidList,* specified within the call to QueBurstSendStart(), are evaluated subsequently by *each* QueBurstSend() operation for determining the target queue of those operations. It is possible to specify QUE_NULL_QUESELECTCODE and QUE_NULL_QIDLIST for *QueSelectCode* and *QidList* respectively. Doing so indicates that all QueBurstSend() calls within the upcoming send-burst will specify a non-null (i.e., valid) *TargetQid*. (Refer to QueBurstSend() for details.)

The *ReadAheadBufSize* parameter is used to specify the size of the buffer used to read ahead network messages. This value may be set to QUE_BURST_DEFAULT_READAHEADSIZE, in which case X•IPC determines a value based on underlying protocol configuration settings. Larger values for this parameter will generally produce higher message throughput, at the cost of additional memory utilization on the instance node.

The *MaxMsgLength* parameter is used to specify the maximum length for all messages in the upcoming send-burst. If a longer message is encountered, an error status is returned and the send-burst is terminated.

The *BlockingOption* specifies the blocking option to be used when enqueuing burst messages during the upcoming send-burst. Note that when QUE_NOWAIT or QUE_TIMEOUT is specified, and a subsequent send-burst message cannot be enqueued (e.g., because of a queue capacity limitation), an error condition results and the send-burst is terminated.

The *ErrorOption* specifies the error reporting callback function to be invoked if an error occurs during any of the QueBurstSend() enqueuing operations within the upcoming send-burst. Enqueuing errors are *not* returned synchronously by the QueBurstSend() operation but instead are reported asynchronously. The *ErrorOption* must specify QUE_CALLBACK(*Function*, *Acb*), where *Function* is called for handling enqueuing errors, and details of enqueuing errors are reported within *Acb*. Included within *Acb* are: the burst message sequence number of the message that was not successfully enqueued, and the cause of the error. Refer to QueBurstSend() for more details.

Error reporting by the QueBurstSendStart() function call is synchronous in nature. The function returns an error code indicating any error encountered while attempting to start a send-burst. The error codes and their interpretations are listed below.

For a more detailed discussion of QueBurstSendStart(), refer to the Advanced Topics chapter in the QueSys/MemSys/SemSys User Guide.

ERRORS

Code	Description
QUE_ER_BADBLOCKOPT	Invalid *BlockingOption* parameter.
QUE_ER_BADERROROPT	Invalid *ErrorOption* parameter.
QUE_ER_BADLENGTH	Invalid *MaxMsgLength* parameter.
QUE_ER_BADQID	Bad Qid in QidList.
QUE_ER_BADQIDLIST	Invalid QidList parameter.
QUE_ER_BADQUESELECTCODE	Invalid QueSelectCode parameter.
QUE_ER_BADREADAHEAD	Invalid ReadAheadBufSize parameter.
QUE_ER_INRECEIVEBURST	User is in a receive-burst.
QUE_ER_INSENDBURST	User already in a send-burst.
QUE_ER_ISFROZEN	A *BlockingOption* of QUE_WAIT or QUE_TIMEOUT was specified after the instance or QueSys was frozen by the calling user.
QUE_ER_NOASYNC	Asynchronous environment not present.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
QUE_ER_SYSERR	Send-burst not started due to system error.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.
XIPCNET_ER_TOOBIG	*MaxMsgLength* exceeds instance's network size limit.

INTERACTIVE COMMAND

SYNTAX

{queburstsendstart|qbsstart} *QueSelectCode QidList MaxMsgLength ReadAheadBufSize BlockingOpt ErrorOption*

ARGUMENTS

QueSelectCode A message dispatch queue select code. (See later in this chapter, under Macros.) The prefix "QUE_Q_" of the queue select code should be omitted, e.g., instead of QUE_Q_ANY, use **any**.

QidList A list of queue Ids

MaxMsgLength The maximum size (in bytes) of messages to be sent in the upcoming send-burst.

ReadAheadBufSiz The size, in bytes, of the read-ahead buffer used by *X♦IPC* to read-ahead messages off the network. This value may be set to **default**, in which case *X♦IPC* uses determines a value based on underlying protocol configuration settings.

BlockingOpt The blocking option to be used when enqueuing messages during the burst. See the Blocking Options discussion in section 2.8.1.4 of the *X♦IPC* Reference

Manual. Only one of the three synchronous values (**wait, nowait** or **timeout**) is permitted

ErrorOption The **callback** option specifying how enqueue error conditions should be reported during the upcoming send-burst. See the Callback Routines discussion in section 2.8.1.3 of the *X♦IPC* Reference Manual. Error information is reported asynchronously within the specified option's associated ACB.

EXAMPLES

```
xipc> queburstsendstart any 0 64 1024 wait callback(cb1, a)
      RetCode = 0

xipc> qbsstart shq 0,1 64 default timeout(15) callback(cb1, a)
      RetCode = 0
```

2.2.6 QueBurstSendStop() - STOP A SEND-BURST

NAME

`QueBurstSendStop()` - Stop a Send-Burst

SYNTAX

```
#include "xipc.h"


XINT
QueBurstSendStop()
```

PARAMETERS

None

RETURNS

Value	Description
RC > 0	Sequence number of last message successfully enqueued.
RC < 0	Error (see error codes below).

DESCRIPTION

QueBurstSendStop() stops a send-burst. QueBurstSendStop() returns the sequence number of the last message that was sent *and* successfully enqueued.
It is not permitted to issue a QueBurstSend() call following a call to QueBurstSendStop() and before a new send-burst has been started.
For a more detailed discussion of QueBurstSendStop(), refer to the Advanced Topics chapter in the QueSys/MemSys/SemSys User Guide.

ERRORS

Code	Description
`QUE_ER_NOTINSENDBURST`	User not in send-burst.
`QUE_ER_NOTLOGGEDIN`	User not logged into instance (User never logged in, was aborted or disconnected).
`XIPCNET_ER_CONNECTLOST`	Connection to instance lost.
`XIPCNET_ER_NETERR`	Network transmission error.
`XIPCNET_ER_SYSERR`	Operating system error.

INTERACTIVE COMMAND

SYNTAX

{queburstsendstop | qbsstop}

ARGUMENTS

None.

EXAMPLE

```
xipc> qbsstop
      SeqNo = 104408
```

2.2.7 QueBurstSendSync() - SYNCHRONIZE A SEND-BURST

NAME

`QueBurstSendSync()` - Synchronize A Send-Burst

SYNTAX

```
#include "xipc.h"


XINT
QueBurstSendSync(Mode)
```

PARAMETERS

Name	Description
Mode	Either QUE_WAIT or QUE_CALLBACK(*UserCallBack, AcbPtr*). See Description below for details.

RETURNS

Value	Description
RC > 0	*Mode* was specified as QUE_WAIT. RC is the burst sequence number of the last send-burst message sent and successfully enqueued.
RC < 0	RC is set as QUE_ER_ASYNC when *Mode* was QUE_CALLBACK() and the operation successfully went async. Otherwise, RC indicates an error situation (see error codes below).

DESCRIPTION

Enqueuing messages in burst mode, using QueBurstSend(), does not provide a per message return code indicating whether sent messages were successfully enqueued. Error reporting during a send-burst, being asynchronous, can suffer from some latency. The situation can arise when an application needs to confirm, periodically, that all messages sent during the current send-burst have been successfully enqueued. The QueBurstSendSync() operation provides such a mechanism.
QueBurstSendSync() operates in one of two modes, depending on the specified value of *Mode*:

- QUE_WAIT - directs QueBurstSendSync() not to return until all messages already in the send-burst have been successfully enqueued. It then returns the sequence number of the last enqueued message.
- QUE_CALLBACK(*UserCallBack, AcbPtr*) - directs QueBurstSendSync() to return its results asynchronously. The advantage of such a mode is that it provides a means for receiving sync-point data about enqueued messages during a send-burst *without* temporarily interrupting the flow of send-burst data.

For a more detailed discussion of QueBurstSendSync(), refer to the Advanced Topics chapter in the QueSys/MemSys/SemSys User Guide.

ERRORS

Code	Description
QUE_ER_ASYNC	*Mode* was QUE_CALLBACK(). Operation has gone async.
QUE_ER_BADSYNCMODE	Invalid *Mode* parameter.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTINSENDBURST	User not in send-burst.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

{queburstsendsync | qbssync} *Mode*

ARGUMENTS

Mode Either **wait** or a **callback** option. See the Advanced Topics section for a discussion of Blocking Options.

EXAMPLE

```
xipc> queburstsendsync  wait
      SeqNo = 104408

xipc> qbssync  callback(cb1, a)
      RetCode = -1097
      Operation has gone Async
```

2.2.8 QueCopy() - COPY PORTION OF MESSAGE TEXT FROM TEXT POOL

NAME

QueCopy() - Copy Portion Of Message Text From Text Pool

SYNTAX

```
#include "xipc.h"


XINT
QueCopy(MsgHdr, Offset, Length, Buffer)

MSGHDR *MsgHdr;
XINT Offset;
XINT Length;
XANY *Buffer;
```

PARAMETERS

Name	Description
MsgHdr	A pointer to a message header. *MsgHdr* refers to a message whose text is recorded in the message text pool.
Offset	The number of bytes beyond the start of the message's text where the QueCopy() operation should commence.
Length	The number of bytes to copy from the message's text, starting at *Offset* bytes into the message.
Buffer	A pointer to a buffer for receiving the copied text.

RETURNS

Value	Description
RC >= 0	Copy successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueCopy() copies a portion of a message's text from the message text pool into a user specified buffer. QueCopy() accesses the message's text using its message header. Unlike QueRead(), QueCopy() does *not* remove the copied text from the text pool and therefore does *not* decrement the associated text block's reference count.
QueCopy() will fail if *MsgHdr* is invalid or if the specified *Offset* and *Length* parameters target an area that is outside the message's actual text space.
QueCopy() can be used for examining the contents of a message in a manner that is *not* sensitive to the instance's physical network location.

ERRORS

Code	Description
QUE_ER_BADBUFFER	*Buffer* is NULL.
QUE_ER_BADTARGET	*Offset* and *Length* designate an invalid message text target area.
QUE_ER_BADTEXT	Invalid *MsgHdr* text pointer.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.
XIPCNET_ER_TOOBIG	Targeted text exceeds instance's size limit.

INTERACTIVE COMMAND

SYNTAX

quecopy *MsgHdr Offset Length*

ARGUMENTS

MsgHdr A one letter message header variable.

Offset Offset from the start of message text.

Length Length of text to copy, or ***** to copy the entire message.

EXAMPLES

```
xipc> queget a hp 0 wait
      RetCode = 0, Qid = 0, Seq# = 1221, Prio = 100, HdrStatus = REMOVED
xipc> quecopy a 0 22
      Text = "Mary had a little lamb"
```

2.2.9 QueCreate() - CREATE A NEW QUEUE

NAME

`QueCreate()` - Create a New Queue

SYNTAX

```
#include "xipc.h"


XINT
QueCreate(Name, LimitMsgs, LimitBytes)

CHAR *Name;
XINT LimitMsgs;
XINT LimitBytes;
```

PARAMETERS

Name	Description
Name	A pointer to a string that contains a symbolic name for publicly identifying the created queue. The *Name* must be null terminated and must not exceed QUE_LEN_XIPCNAME characters. If *Name* is QUE_PRIVATE then a private queue is created. Duplicate queue names (other than QUE_PRIVATE) are not allowed.
LimitMsgs	The maximum number of messages that can reside on the queue at any one time. A value of QUE_NOLIMIT indicates that no limit is to be enforced on the number of messages allowed on the queue.
LimitBytes	The maximum number of message text bytes that can reside on the queue at any one time. A value of QUE_NOLIMIT indicates that no limit is to be enforced on the number of message text bytes allowed on the queue.

RETURNS

Value	Description
RC >= 0	Create successful. RC is Queue ID (Qid). Qid is to be used in all subsequent QueSys calls that refer to this queue.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueCreate() creates a new queue in QueSys. *Name* is used for publicly identifying the new queue. A *Name* of QUE_PRIVATE directs QueSys to create a private queue (i.e., having no public name). The function returns the Qid of the created queue.
The queue is created with the capacity to hold up to a maximum of *LimitMsgs* messages and *LimitBytes* bytes, whichever limit occurs first. Setting both limits to QUE_NOLIMIT creates a

queue which has no enforced size limitation and which will accept messages until overall instance limitations are encountered (e.g., headers, text pool space, etc.).

ERRORS

Code	Description
QUE_ER_BADLIMIT	Invalid *LimitMsgs* or *LimitBytes* parameter.
QUE_ER_BADQUENAME	Invalid *Name* parameter.
QUE_ER_CAPACITY_HEADER	QueSys header table full.
QUE_ER_CAPACITY_NODE	QueSys node table full.
QUE_ER_CAPACITY_TABLE	Queue table full.
QUE_ER_DUPLICATE	Queue with *Name* already exists.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

quecreate {*Name* | @PRIVATE} *LimitMsgs LimitBytes*

ARGUMENTS

Name Name of new queue (or, if @PRIVATE, a private queue indicator).

LimitMsgs Either maximum number of messages in queue or **nolimit**.

LimitBytes Either maximum number of text bytes in queue or **nolimit**.

EXAMPLES

```
xipc> quecreate ServerQueue nolimit 10000
      Qid = 0
```

2.2.10 QueDelete() - DELETE A QUEUE

NAME

`QueDelete()` - Delete a Queue

SYNTAX

```
#include "xipc.h"


XINT
QueDelete(Qid)

XINT Qid;
```

PARAMETERS

Name	Description
`Qid`	The Queue ID of the queue to be deleted. *`Qid`* was obtained by the user via QueCreate() or QueAccess() function calls.

RETURNS

Value	Description
RC >= 0	Delete successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueDelete() deletes the Queue identified by *`Qid`* from QueSys. QueDelete() will fail if the specified queue is not empty, or if any users are waiting to dispatch or retrieve messages via queue *`Qid`*.

ERRORS

Code	Description
QUE_ER_BADQID	No queue with *`Qid`*.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTEMPTY	The queue is not empty.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
QUE_ER_WAITEDON	A user is waiting for a message on *`Qid`*.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

quedelete *Qid*

ARGUMENTS

Qid Queue Id.

EXAMPLES

```
xipc> quedelete 5
      RetCode = 0
```

2.2.11 QueDestroy() - DESTROY A QUEUE

NAME

QueDestroy() - Destroy a Queue

SYNTAX

```
#include "xipc.h"


XINT
QueDestroy(Qid)

XINT Qid;
```

PARAMETERS

Name	Description
Qid	The Queue ID of the queue to be destroyed. *Qid* was obtained by the user via QueCreate() or QueAccess() function calls.

RETURNS

Value	Description
RC >= 0	Destroy successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueDestroy() removes the Queue identified by *Qid* from QueSys regardless of whether it has messages on it, or whether other users are waiting to send or receive messages via the queue. Messages residing on the queue are silently destroyed, reducing the text block count to zero. Users blocked on QueSys calls involving queue *Qid* (such as QueSend(), QueReceive(), QuePut() or QueGet()) are interrupted and are returned an error code of QUE_ER_DESTROYED indicating the removal of queue *Qid* from QueSys.

ERRORS

Code	Description
QUE_ER_BADQID	No queue with *Qid*.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

quedestroy *Qid*

ARGUMENTS

Qid Queue Id.

EXAMPLES

```
xipc> quedestroy 5
      RetCode = 0
```

2.2.12 QueFreeze() - FREEZE QUESYS

NAME

`QueFreeze()` - Freeze QueSys

SYNTAX

```
#include "xipc.h"


XINT
QueFreeze()
```

PARAMETERS

None.

RETURNS

Value	Description
RC >= 0	QueFreeze successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueFreeze() freezes all QueSys activity occurring within the logged in instance, and gives the calling user exclusive access to all QueSys functionality. QueSys remains frozen until a QueUnfreeze(), XipcUnfreeze() or a XipcLogout() function call is issued.
QueFreeze() prevents all other users, working within the QueSys, from proceeding with QueSys operations - until a bracketing QueUnfreeze(), XipcUnfreeze() or XipcLogout() is issued. The subsystem should therefore be kept frozen for as short a period of time as possible.

ERRORS

Code	Description
`QUE_ER_NOSUBSYSTEM`	QueSys is not configured in the instance.
`QUE_ER_NOTLOGGEDIN`	User not logged into instance (User never logged in, was aborted or disconnected).
`QUE_ER_ISFROZEN`	Calling user has already frozen QueSys.
`XIPCNET_ER_CONNECTLOST`	Connection to instance lost.
`XIPCNET_ER_NETERR`	Network transmission error.
`XIPCNET_ER_SYSERR`	Operating system error.

INTERACTIVE COMMAND

SYNTAX

quefreeze

ARGUMENTS

None.

EXAMPLES

```
xipc> quefreeze
      RetCode = 0
```

2.2.13 QueGet() - GET A MESSAGE HEADER FROM A QUEUE

NAME

`QueGet()` - Get a Message Header From a Queue

SYNTAX

```
#include "xipc.h"


XINT
QueGet(MsgHdr, QueSelectCode, QidList, RetPrio, RetQid,
       Options)

MSGHDR *MsgHdr;
XINT QueSelectCode;
QIDLIST QidList;
XINT *RetPrio;
XINT *RetQid;
... Options;
```

PARAMETERS

Name	Description
MsgHdr	A pointer to a message header. *MsgHdr* is returned with the header information of the gotten message.
QueSelectCode	A code indicating the selection criteria to be used in determining the gotten message of the QueGet() operation. The selected message is taken from one of the Qids in *QidList*. The possible values for *QueSelectCode* are listed in Appendix B.
QidList	A list of Qids, possibly specified within Message Select Code macros, to be used in specifying candidate messages for consideration by the QueGet() operation. QueGet() selects one of the candidate messages based on the value of *QueSelectCode*. A QIDLIST is constructed using QueList() or QueListBuild() and is updated using QueListAdd(). A pointer to a QIDLIST (type PQIDLIST) may be passed as well.
RetPrio	A pointer to a variable that gets set by QueGet(), upon successful return, with the priority of the retrieved message; or NULL if no return value is desired.
RetQid	A pointer to a variable that gets assigned by QueGet() upon its return; or NULL if no return value is desired. Successful QueGet() operations (RC >= 0) return with **RetQid* equal to the Qid of the selected source queue (from within QidList) that provided the gotten message. Cancelled QueGet() operations having RC = QUE_ER_DESTROYED or QUE_ER_PURGED, return with **RetQid* equal to the destroyed or purged Qid. Failed calls with RC = QUE_ER_BADQID return with **RetQid* equal to the invalid Qid. **RetQid* is otherwise undefined.

Options

The *Options* parameter is of the form:

[QUE_NOREMOVE |] *BlockOpt*

The QUE_NOREMOVE flag (placed to the left of the blocking option) is optional. When specified, the accessed message header is *not* dequeued. Rather, a fully functional copy of the message header is retrieved. *BlockOpt* specifies the blocking option. See Appendix A, Using Blocking X•IPC Functions, for a description of *BlockOpt*.

RETURNS

Value	Description
RC >= 0	Get successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueGet() attempts to get (and possibly remove) a QueSys message header from one of the Qids in *QidList*. The determination of which message header is gotten is based on each Qid's Message Select Code as listed in *QidList*, in conjunction with the value of *QueSelectCode*. QueGet() sets *MsgHdr* with header information from the retrieved message header . **RetPrio* gets assigned with the retrieved message's priority.

It is acceptable to have a null *RetQid* or *RetPrio* argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired. In addition, QueGet() can specify whether the returned *RetPrio* should be populated with the *Priority* of the retrieved message or the *Sequence number* of the retrieved message. This is accomplished by using one of the two optional flags, QUE_RETPRIO or QUE_RETSEQ, which determine which value is returned with the message. The default value is QUE_RETPRIO, in order to preserve backward compatibility.

The retrieved message is removed from the queue unless the QUE_NOREMOVE option is specified. In that case, a fully functional copy of the gotten message header is returned, and the actual header remains on the queue. In either case, the message text (if any) remains untouched, in the message text pool. The returned message header copy may subsequently be used by QueBrowse() as a reference point for browsing a queue's messages. Alternatively, it can be used via QueRemove() to dequeue the actual header.

QueGet() is given the potential to block or complete asynchronously by setting *BlockOpt* appropriately. The operation will block or complete asynchronously if either all the listed queues are empty, or they contain messages not matching their respective Message Select Codes. The QueGet() operation will complete when the reason for not completing is removed, usually by another user's actions. Specifically:

- ☐ Another user calling QueSend() or QuePut() to place a message on one of the involved queues.

See Appendix A, Using Blocking X•IPC Functions, for a description of how to use the blocking options.

Message headers removed via QueGet() from one queue can be placed via QuePut() onto another queue. Message transfer can thus be accomplished *without* any message text copying.

The MSGHDR data structure is defined as follows:

```
typedef struct _MSGHDR
{
    XINT GetQid;                    /* Last Qid msg was on */
    XINT HdrStatus;                     /* Rmvd or Not Rmvd, etc */
    XINT Priority;                      /* Message's priority */
    XINT SeqNum;                    /* Msg sequence # within queue */
    XINT TimeVal;                       /* Msg sequence number within QueSys */
    XINT Size;                          /* Numb. of bytes in msg */
    XINT TextOffset;                /* Offset of msg's text in text-pool */
    XINT Uid;                           /* The User-Id of user that sent msg */
    CHAR Data[MSGHDR_DATASIZE];         /* User data field */
}
MSGHDR;
```

ERRORS

Code	Description
QUE_ER_ASYNC	Operation is being performed asynchronously.
QUE_ER_ASYNCABORT	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
QUE_ER_BADBLOCKOPT	Invalid *BlockOpt*.
QUE_ER_BADMSGSELECTCODE	Invalid MsgSelectCode within *QidList*.
QUE_ER_BADOPTION	Invalid *Options* parameter.
QUE_ER_BADQID	Bad Qid in QidList (= **QidPtr*).
QUE_ER_BADQIDLIST	Invalid *QidList* parameter.
QUE_ER_BADQUESELECTCODE	Invalid *QueSelectCode* parameter.
QUE_ER_CAPACITY_ASYNC_USER	QueSys async user table full.
QUE_ER_CAPACITY_NODE	QueSys node table full.
QUE_ER_DESTROYED	Another user destroyed a queue that the blocked QueGet() call was waiting on. The blocked QueGet() operation was cancelled. No message was gotten.
QUE_ER_INTERRUPT	Operation was interrupted.
QUE_ER_ISFROZEN	A *BlockOpt* of QUE_WAIT or QUE_TIMEOUT() was specified after the instance was frozen by the calling user.
QUE_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
QUE_ER_NOWAIT	*BlockOpt* of QUE_NOWAIT was specified and request was not immediately satisfied.
QUE_ER_PURGED	Another user purged a queue that the blocked QueGet() call was waiting on. The blocked QueGet() operation was cancelled. No message was gotten.
QUE_ER_TIMEOUT	The blocked QueGet() operation timed out.

XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

queget *MsgHdr QueSelectCode QidList* [**noremove,**] [**retprio**, | **retseq,**] *BlockingOpt*

ARGUMENTS

MsgHdr — A one letter message header variable.

QueSelectCode — A message retrieval queue select code. (This is presented in the QueSys Functions and Macros chapter, under Macros.) The prefix "QUE_Q_" of the queue select code should be omitted; e.g., instead of QUE_Q_EA, use **ea**.

QidList — A list of Queue Ids, possibly specified with message select codes. (This is presented in the QueSys Functions and Macros chapter, under Macros.) The prefix "QUE_M_" of the message select code should be omitted; e.g., instead of QUE_M_HP(1), use **hp(1).**

BlockingOpt — See the Blocking Options discussion in the xipc command (Interactive Command Processor) section at the beginning of this Manual.

EXAMPLES

```
xipc> queget a ea 0 wait
      RetCode = 0, Qid = 0, Seq# = 1221, Prio = 100, HdrStatus = REMOVED


xipc> queget c lnq hp(0),hp(1) wait
      RetCode = 0, Qid = 1, Seq# = 21221, Prio = 1100, HdrStatus = REMOVED


xipc> queget e hp 1,2,4,8 noremove,nowait
      RetCode = 0, Qid = 4, Seq# = 212, Prio = 1101, HdrStatus = NOT-REMOVED
```

2.2.14 QueInfoQue() - GET QUEUE INFORMATION

NAME

`QueInfoQue()` - Get Queue Information

SYNTAX

```
#include "xipc.h"


XINT
QueInfoQue(Qid, InfoQue)

XINT Qid;
QUEINFOQUE *InfoQue;
```

PARAMETERS

Name	Description
`Qid`	The queue ID of the queue whose information is desired, or `QUE_INFO_FIRST,` or `QUE_INFO_NEXT(`*`Qid`*`).` *Qid* can be obtained via QueCreate() or QueAccess() function calls.
`InfoQue`	Pointer to a structure of type `QUEINFOQUE`, into which the queue information will be copied.

RETURNS

Value	Description
RC >= 0	Successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueInfoQue() fills the specified structure with information about the queue identified by *Qid*. The Qid argument can be specified as one of the following:

- *`Qid`* - a queue id identifying a specific queue
- `QUE_INFO_FIRST` - identifies the first valid queue id
- `QUE_INFO_NEXT(`*`Qid`*`)` - identifies the next valid queue id, following *`Qid`*.

A program reviewing the status of all queues within an instance should call QueInfoQue() specifying `QUE_INFO_FIRST,` followed by repeated calls to the function specifying `QUE_INFO_NEXT` until the `QUE_ER_NOMORE` error code is returned.

Each QueSys queue has a Wait List (*WList*) of information associated with it; the elements comprising a queue's WList depend on the mix of blocked operations occurring at the time of the QueInfoQue call:

- A WList element exists for every blocked QuePut() or QueSend() operation targeting the subject queue. The list element identifies the nature of the blocked QuePut or QueSend operation.

- A WList element exists for every blocked QueGet() or QueReceive() operation involving the subject queue. The list element identifies the details of the blocked QueGet() or QueReceive() operation.

The QUEINFOQUE data structure is as follows:

```
/*
 * The QUEINFOQUE structure is used for retrieving status information
 * about a particular QueSys message queue. QueInfoQue() fills the
 * structure with the data about the Qid it is passed.
 */
typedef struct _QUEINFOQUE
{
    XINT Qid;
    XINT CreateTime;                /* Time queue was created */
    XINT CreateUid;                 /* The Uid who created it */
    XINT LastUid;                       /* Last Uid to use queue */
    LBITS QueType;                      /*  - Not Used - */
    XINT LimitMessages;             /* Max message capacity */
    XINT LimitBytes;                /* Max byte capacity */
    XINT CountMessages;             /* Current number of msgs */
    XINT CountBytes;                /* Current number of bytes */
    XINT CountIn;                       /* Number msgs entered que */
    XINT CountOut;                      /* Number msgs exited que */
    XINT LastUidGet;                /* Last Uid to put a msg */
    XINT LastUidPut;                /* Last Uid to get a msg */
    XINT SpoolFlag;                 /* Spooling: ON or OFF */
    XINT SpoolMessages;             /* Number msgs spooled */
    XINT SpoolBytes;                /* Number bytes spooled */
    XINT WListTotalLength;
    XINT WListOffset;
    XINT WListLength;
    QUE_QUEWLISTITEM WList[QUE_LEN_INFOLIST];
    CHAR SpoolFileName[QUE_LEN_PATHNAME+1];
    CHAR Name[QUE_LEN_XIPCNAME + 1]; /* Queue name */
}
QUEINFOQUE;
```

where:

WListTotalLength returns with the total internal length of the WList for this queue.

WListOffset is set by the user, prior to the QueInfoQue() function call, to specify the portion of the WList that should be returned (i.e. what offset to start from).

WListLength returns with the length of the WList portion returned by the current call to QueInfoQue(). More specifically, *WListLength* is the number of elements returned in the *WList* array. *WListLength* will be between 0 and QUE_LEN_INFOLIST.

WList is an array of list elements, where each element is of type QUE_QUEWLISTITEM. The QUE_QUEWLISTITEM data type is defined in quepubd.h. The data structure follows:

```
typedef struct _QUE_QUEWLISTITEM
{
    XINT  OpCode;               /* PUT or GET */
    union
    {
        struct
        {
            XINT Uid;           /* User blocked */
            XINT MsgSize;       /* Putting Msg */
            XINT MsgPrio;       /* Msg Priority */
        }
```

```
        Put;

        struct
        {
            XINT Uid;           /* User blocked */
            XINT MsgSelCode; /* Getting Msg */
            XINT Parm1;
            XINT Parm2;
        }
        Get;
    }
    u;
}
QUE_QUEWLISTITEM;
```

A call to QueInfoQue() should be preceded by the setting of the *WListOffset* field of the QUEINFOQUE structure to an appropriate value.

For a full example and more complete information on using the Info functions, refer to the Advanced Topics chapter of the *X♦IPC* User Guide.

ERRORS

Code	Description
QUE_ER_ASYNC	Operation is being performed asynchronously.
QUE_ER_ASYNCABORT	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
QUE_ER_BADBLOCKOPT	Invalid *BlockOpt*.
QUE_ER_BADMSGSELECTCODE	Invalid MsgSelectCode within *QidList*.
QUE_ER_BADOPTION	Invalid *Options* parameter.
QUE_ER_BADQID	Bad Qid in QidList (= **QidPtr*).
QUE_ER_BADQIDLIST	Invalid *QidList* parameter.
QUE_ER_BADQUESELECTCODE	Invalid *QueSelectCode* parameter.
QUE_ER_CAPACITY	QueSys internal system capacity error.
QUE_ER_DESTROYED	Another user destroyed a queue that the blocked QueGet() call was waiting on. The blocked QueGet() operation was cancelled. No message was gotten.
QUE_ER_INTERRUPT	Operation was interrupted.
QUE_ER_ISFROZEN	A *BlockOpt* of QUE_WAIT or QUE_TIMEOUT() was specified after the instance was frozen by the calling user.
QUE_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.
QUE_ER_NOMORE	No more queues.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).

QUE_ER_NOWAIT	*BlockOpt* of QUE_NOWAIT was specified and request was not immediately satisfied.
QUE_ER_PURGED	Another user purged a queue that the blocked QueGet() call was waiting on. The blocked QueGet() operation was cancelled. No message was gotten.
QUE_ER_TIMEOUT	The blocked QueGet() operation timed out.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

queinfoque *QueId* | **first** | **next(***QueId***)**

ARGUMENTS

QueId Print info on the **first** queue, the queue with Qid *QueId* or the **next** higher queue.

EXAMPLES

```
xipc> queinfoque 5
      Name: 'ServerQue'
      Message limit: 100     Bytes Limit: 10000
      . . .
```

2.2.15 QueInfoSys() - GET SUBSYSTEM INFORMATION

NAME

`QueInfoSys()` - Get Subsystem Information

SYNTAX

```
#include "xipc.h"


XINT
QueInfoSys(InfoSys)

QUEINFOSYS *InfoSys;
```

PARAMETERS

Name	Description
`InfoSys`	Pointer to a structure of type `QUEINFOSYS`, into which subsystem information will be copied.

RETURNS

Value	Description
RC >= 0	Successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueInfoSys() fills the specified structure with information about the QueSys of the instance that the user is currently logged into.
The QueSys subsystem has a Wait List (*WList*) of information associated with it; the elements comprising the WList are as follows:

- A WList element exists for every blocked QueWrite() operation occurring in the subsystem at the time of the QueInfoSys() call.

The QUEINFOSYS data structure is as follows:

```
/*
 * The QUEINFOSYS structure is used for retrieving status information
 * about the QueSys instance. QueInfoSys() fills the structure with the
 * data about the instance.
 */

typedef struct _QUEINFOSYS              /* system information */
{
    XINT MaxUsers;                      /* Max configured users */
    XINT CurUsers;                      /* Number of current users */
    XINT MaxQueues;                 /* Max configured queues */
    XINT CurQueues;                 /* Number of current queues */
    XINT MaxNodes;                      /* Max configured nodes */
    XINT FreeNCnt;                      /* Current available nodes */
    XINT MaxHeaders;                /* Max configured headers */
    XINT FreeHCnt;                      /* Current available hdrs */
    XINT SplTickSizeBytes;              /* Configured spool tick value */
    XINT MsgPoolSizeBytes;              /* Configured text pool size */
    XINT MsgTickSize;                   /* Configured text tick size */
    XINT MsgPoolTotalAvail;             /* Free text pool space */
    XINT MsgPoolLargestBlk;             /* Largest contig block */
    XINT MsgPoolMaxPosBlks;             /* Max possible tick blocks */
    XINT MsgPoolTotalBlks;              /* Number allocated blocks */
    XINT WListTotalLength;
    XINT WListOffset;
    XINT WListLength;
    QUE_SYSWLISTITEM WList[QUE_LEN_INFOLIST];
    CHAR Name[QUE_LEN_PATHNAME + 1]; /* InstanceFileName */
}
QUEINFOSYS;
```

where:

WListTotalLength returns with the total internal length of the WList.

WListOffset is set by the user, prior to the QueInfoSys() function call, to specify the portion of the WList that should be returned (i.e. what offset to start from).

WListLength returns with the length of the WList portion returned by the current call to QueInfoSys(). More specifically, *WListLength* is the number of elements returned in the *WList* array. *WListLength* will be between 0 and `QUE_LEN_INFOLIST`.

WList is an array of list elements, where each element is of type `QUE_SYSWLISTITEM`. The `QUE_SYSWLISTITEM` data type is defined in quepubd.h. The data structure follows:

```
typedef struct _QUE_SYSWLISTITEM
{

    XINT Uid;               /* User Blked */
    XINT MsgSize;           /* Write size */
}
QUE_SYSWLISTITEM;
```

A call to QueInfoSys() should be preceded by the setting of the *WListOffset* field of the QUEINFOSYS structure to an appropriate value.

For a full example and more complete information on using the Info functions, refer to the Advanced Topics chapter of the X•IPC User Guide.

ERRORS

Code	Description
QUE_ER_BADLISTOFFSET	Invalid offset value specified.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

queinfosys

ARGUMENTS

None.

EXAMPLES

```
xipc> queinfosys
      Configuration:  '/usr/config'
      .......  Maximum  Current
      Users:        60       11
      .
      .
      .
```

2.2.16 QueInfoUser() - GET QUESYS USER INFORMATION

NAME

QueInfoUser() - Get User Information

SYNTAX

```
#include "xipc.h"


XINT
QueInfoUser(Uid, InfoUser)

XINT Uid;
QUEINFOUSER *InfoUser;
```

PARAMETERS

Name	Description
Uid	The user ID of the user whose information is desired, or QUE_INFO_FIRST, or QUE_INFO_NEXT(*Uid*). *Uid* may be an asynchronous Uid (AUid).
InfoUser	Pointer to a structure of type QUEINFOUSER, into which the user information will be copied.

RETURNS

Value	Description
RC >= 0	Successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueInfoUser() fills the specified structure with information about the user identified by *Uid*. The Uid argument can be specified as one of the following:

- *Uid* - an integer user ID identifying a specific user
- QUE_INFO_FIRST - identifies the first valid user ID within the instance
- QUE_INFO_NEXT(*Uid*) - identifies the next valid user ID, following *Uid*.

A program reviewing the status of all users currently within QueSys would call QueInfoUser() specifying QUE_INFO_FIRST, followed by repeated calls to the function specifying QUE_INFO_NEXT until the QUE_ER_NOMORE error code is returned.

Each QueSys user has a Wait List (*WList*) of information associated with it.

A user can be blocked on one of three QueSys operations: QuePut(), QueGet() or QueWrite(). The elements comprising the WList depend on the operation involved:

- During a blocked QuePut() operation, the WList identifies the list of Qids targeted by the blocked QuePut() (or QueSend()) call.
- During a blocked QueGet() operation, the WList identifies the list of Qids, and their respective Message Select Criteria, targeted by the blocked QueGet() (or QueReceive()) call.

- During a blocked QueWrite() operation, the WList is a single element list. The list element identifies the nature of the blocked QueWrite() (or QueSend()) operation.

The QUEINFOUSER data structure is as follows:

```
/*
 * The QUEINFOUSER structure is used for retrieving status information
 * about a particular QueSys user. QueInfoUser() fills the structure
 * with the data about the Uid it is passed.
 */

typedef struct _QUEINFOUSER
{
        XINT Uid;
        XINT Pid;                               /* Process ID of user */
        TID  Tid;                               /* Thread ID of user */
        XINT LoginTime;                 /* Time of login to QueSys */
        XINT TimeOut;                           /* Remaining timeout secs */
        XINT WaitType;                          /* One of:  QUE_BLOCKEDWRITE,
QUE_BLOCKEDPUT,                                                         *
QUE_BLOCKEDGET or QUE_USER_NOTWAITING
                                                        */
        XINT CountPut;                          /* Number of msgs put */
        XINT CountGet;                          /* Number of msgs gotten */
        XINT LastQidPut;                /* Last Qid msg was put on */
        XINT LastQidGet;                /* Last Qid msg taken from */
        XINT WListTotalLength;
        XINT WListOffset;
        XINT WListLength;
        QUE_USERWLISTITEM WList[QUE_LEN_INFOLIST];
        CHAR Name[QUE_LEN_XIPCNAME + 1]; /* User login name /
        CHAR NetLoc[XIPC_LEN_NETLOC + 1];       /* Name of Client Node */
}
QUEINFOUSER;
```

where:

WListTotalLength returns with the total internal length of the WList for the specified user.

WListOffset is set by the user, prior to the QueInfoUser() function call, to specify the portion of the WList that should be returned (i.e. what offset to start from).

WListLength returns with the length of the WList portion returned by the current call to QueInfoUser(). More specifically, *WListLength* is the number of elements returned in the *WList* array. *WListLength* will be between 0 and `QUE_LEN_INFOLIST`.

WList is an array of list elements, where each element is of type `QUE_USERWLISTITEM`. The `QUE_USERWLISTITEM` data type is defined in quepubd.h. The data structure follows:

```
typedef struct _QUE_USERWLISTITEM
{
    XINT  OpCode;               /* PUT, GET or WRITE */
```

```
    union
    {
        struct
        {
            XINT Qid;           /* Que Blocked */
            XINT MsgSize;       /* Putting Msg */
            XINT MsgPrio;       /* Msg Priority */
        }
        Put;

        struct
        {
            XINT Qid;           /* Que Blocked */
            XINT MsgSelCode; /* Getting Msg */
            XINT Parm1;
            XINT Parm2;
        }
        Get;
        struct
        {
            XINT MsgSize;       /* Write Blocked */
        }
        Write;
    }
    u;
}
QUE_USERWLISTITEM;
```

A call to QueInfoUser() should be preceded by the setting of the *WListOffset* field of the QUEINFOUSER structure to an appropriate value.
For a full example and more complete information on using the Info functions, refer to the Advanced Topics chapter of the X♦IPC User Guide.

ERRORS

Code	Description
QUE_ER_BADUID	No user with specified *Uid*.
QUE_ER_BADLISTOFFSET	Invalid offset value specified.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
QUE_ER_NOMORE	No more user entries.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

queinfouser *UserId* | **first** | **next(***UserId***)** | **all**

ARGUMENTS

UserId Print info on the **first** user, the user with Uid *Uid* or the **next** higher user.

QueId

EXAMPLES

```
xipc> queinfouser 9
      Name: 'QueueServer'  Pid: 241  Tid: 0
      Login Time:...
      .
      .
      .
```

2.2.17 QueList(), QueListBuild - BUILD LISTS OF QIDS

NAME

QueList() - Create a One-Time List of Qids

QueListBuild() - Build a Reusable List of Qids

SYNTAX

```
#include "xipc.h"


PQIDLIST
QueList(QidListElement1, QidListElement2, ..., QUE_EOL)

XINT QidListElement1;
XINT QidListElement2;
...

PQIDLIST
QueListBuild(QidList, QidListElement1, QidListElement2, ...,
QUE_EOL)

QIDLIST QidList;
XINT QidListElement1;
XINT QidListElement2;
...
```

PARAMETERS

Name	Description
QidList	An area to contain the resultant QIDLIST. A pointer to a QIDLIST (type PQIDLIST) may be passed as well.
QidListElement1,QidListElement2,	The components of the QIDLIST to be built. Each element is either a Qid or a Message Select Code macro applied to a Qid. QUE_EOL must be used to mark the end of the argument list.

RETURNS

Value	Description
RC != NULL	A pointer to the created QIDLIST. For QueListBuild()it is a pointer to the *QidList* specified as an argument. For QueList() it is a pointer to an internal *QidList*.
RC == NULL	*QidList* exceeded QUE_LEN_QIDLIST elements.

DESCRIPTION

These functions are used for building QIDLISTs in a format acceptable by QuePut(), QueGet(), QueSend() and QueReceive(). `QUE_EOL` must be the last argument to QueListBuild() and QueList(). QueListBuild() builds the list in the area specified by *QidList*. QueList() creates the list in an internal static area, and can therefore be safely used only once. The elements specified to QueList() or QueListBuild() are defined as follows, depending on the function call which will use the QIDLIST:

- *Message Dispatch - QuePut() and QueSend().*

  The elements that go into QIDLISTs to be used for message dispatch operations are the Qids of the queues to be targeted by the QuePut() or QueSend() operation.
  For example, the construction of a QIDLIST for a message dispatch operation targeting queues *QidA*, *QidB* and *QidC* would be:

```
QueListBuild(QidList1, QidA, QidB, QidC, QUE_EOL);
```

  The constructed QIDLIST (in this case *QidList1*) could then be used within a QuePut() as follows:

```
QuePut(&MsgHdr, QUE_Q_SHQ, QidList1, Priority, &RetQid, QUE_WAIT);
```

  The above QuePut() would place *MsgHdr* onto the shortest of the three queues represented by *QidA*, *QidB* and *QidC*. In fact the very same QIDLIST could then be reused:

```
QuePut(&MsgHdr, QUE_Q_LPQ, QidList1, Priority, &RetQid, QUE_WAIT);
```

  This QuePut() would select as its target the queue having the lowest priority message among *QidA*, *QidB* and *QidC*.

  The same rules and usage apply to QueSend().

  Note that when QueList() is used, it is usually embedded directly into the called function's sequence of arguments. The following example is identical to the previous QuePut() statement:

```
QuePut(&MsgHdr, QUE_Q_LPQ, QueList(QidA, QidB, QidC, QUE_EOL),
      Priority, &RetQid, QUE_WAIT);
```

- *Message Retrieval - QueGet() and QueReceive().*

  Building a QIDLIST for a QueGet() or a QueReceive() operation is slightly more involved. For these functions the QIDLIST serves two purposes:

  - Presenting the list of source Qids to consider for retrieving a message from.
  - Identifying a "candidate message" from each of the listed Qids.

  As such, the elements are usually Message Select Code macros, as will now be demonstrated. Consider the following example:

```
QueListBuild(QidList2,
            QUE_M_HP(QidA),
            QUE_M_HP(QidB),
            QUE_M_HP(QidC),
            QUE_EOL);
```

  This call builds *QidList2* so that it can be used to identify the highest priority messages from each of the queues *QidA*, *QidB* and *QidC*. *QidList2* can then be used as follows:

```
QueGet(&MsgHdr, QUE_Q_LNQ, QidList2, &Priority, &RetQid, QUE_WAIT);
```

This call retrieves the candidate message from the longest of the three queues. The net effect is the retrieval of the highest priority message from the longest of the three queues *QidA*, *QidB* and *QidC*.

QIDLISTs created using QueListBuild() can be reused for different objectives. For example, *QidList2* as created in the previous example can be used again as follows:

```
QueGet(&MsgHdr, QUE_Q_EA, QidList2, &Priority, &RetQid, QUE_WAIT);
```

This call retrieves the oldest ("earliest arrived") of the highest priority messages residing on the three source queues.

Sections 2.3.1 and 2.3.2 provide a complete and detailed list of the available Message Select Code and Queue Select Code macros.
Refer to the Appendix "Using Message Select Codes and Queue Select Codes" for a thorough description of the 'hows and whens' of working with Message Select Codes and Queue Select Codes.

ERRORS
None.

2.2.18 QueListAdd(), QueListRemove() – UPDATE LIST OF QIDS

NAME

`QueListAdd()` - Add to a List of Qids

`QueListRemove()` - Remove from a List of Qids

SYNTAX

```
#include "xipc.h"

PQIDLIST
QueListAdd(QidList, QidListElement1, QidListElement2, ...,
QUE_EOL)

QIDLIST QidList;
XINT QidListElement1;
XINT QidListElement2;
...


PQIDLIST
QueListRemove(QidList, QidListElement1, QidListElement2,...,
QUE_EOL)

QIDLIST QidList;
XINT QidListElement1;
XINT QidListElement2;
...
```

PARAMETERS

Name	Description
QidList	The QIDLIST to be updated. A pointer to a QIDLIST (type PQIDLIST) may be passed as well.
QidListElement1,QidListElement2,	The components to be added to or removed from the QIDLIST. Each element is either a Qid or a Message Select Code macro applied to a Qid. `QUE_EOL` must be used to mark the end of the argument list.

RETURNS

Value	Description
RC != `NULL`	A pointer to the updated QIDLIST specified as an argument..
RC == `NULL`	The operation failed. The *QidList* specified as an argument remains unchanged.

DESCRIPTION

These functions are used for modifying QIDLISTs by adding or removing elements. The elemnts specified to QueListAdd() or QueList Remove() may be plain Qids, or Qids with Message Selecte Code macros applied, or a combination of both. QUE_EOL must be the last argument to QueListAdd() and QueListRemove ().

QueListAdd() adds elements to an existing QIDLST. The new elements are added at the end of the specified QidList. If the number of elements being added, plus the current number of elements in the QIDLIST, exceeds QUE_LEN_QIDLIST, then the operation fails, NULL is returned and Qidlist remains unchanged.

QueListRemove() removes elements from an existing QIDLIST. QueList Remove() behaves differently for elements which are plain Qids than for elements which are Qids with Message Select Code macros applied.

If a plain Qid is specified, all elements of the QidList containing the Qid are removed. If there are no such element, no change is made to the QidList, but the operation still succeeds.

If a Qid with Message Select Code macro is specified, it must match an element of the QidList exactly, and then that element is removed. If it does not match an element of the QidList, the operation fails.

If the operation succeeds, a poitner to the modified argument QidList is returned, otherwise NULL is returned and the argument QidList remains unchanged.

ERRORS

None.

2.2.19 QueListCount() - GET NUMBER OF ELEMENTS IN A LIST OF QIDS

NAME

`QueListCount()` - Get Number of Elements in a List of Qids

SYNTAX

```
#include "xipc.h"

XINT
QueListCount(QidList)

QIDLIST QidList;
```

PARAMETERS

Name	Description
`QidList`	A QIDLIST or a pointer to a QIDLIST (type PQIDLIST).

RETURNS

Value	Description
RC < 0	The QIDLIST is invalid..
RC > = 0	Number of elements in QidList.

DESCRIPTION

QueListCount() is used for determining the number of elements contained in QIDLIST.

ERRORS

None.

2.2.20 QueMsgHdrDup() - CREATE COPY OF MESSAGE HEADER

NAME

QueMsgHdrDup() - Create Copy of a Message Header

SYNTAX

```
#include "xipc.h"

QueMsgHdrDup(ExistingMsgHdr, NewMsgHdr)
MSGHDR *ExistingMsgHdr;
MSGHDR *NewMsgHdr
```

PARAMETERS

Name	Description
ExistingMsgHdr	A pointer to the message header to be duplicated. *ExisitngMsgHdr* refers to a message whose text is in the Message Text Pool (whether or not the header had been gotten using QUE_NOREMOVE)..
NewMsgHdr	A pointer to the new message header.

RETURNS

Value	Description
RC >= 0	QueMsgHdrDup successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueMsgHdrDup() allows a user to create a copy of a MSGHDR. The critical aspect of this function is that it does not actually copy the text data, but rather internally increments the affected text-pool block's reference count by one. The function is passed a pointer to two MSGHDRs, *ExistingMsgHdr* and *NewMsgHdr*.

Following the call, both MSGHDRs reference the same message text-pool block.

ERRORS

Code	Description
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.
XIPCNET_ER_TOOBIG	Message text exceeds instance's size limit.

INTERACTIVE COMMAND

SYNTAX

quemsghdrdup *ExistingMsgHdr NewMsgHdr*

ARGUMENTS

ExistingMsgHdr A one letter message header variable containing the existing message header to copy.

NewMsgHdr A one letter message header variable which will be assigned the duplicated message header.

EXAMPLES

```
xipc> quemsghdrdup a b
      Qid = 2, Seq# = 146, Prio = 100, Uid = 10, HdrStatus = DUPLICATED
```

2.2.21 QuePointer() - GET POINTER TO A MESSAGE'S TEXT

NAME

`QuePointer()` - Get Pointer to a Message's Text

SYNTAX

```
#include "xipc.h"


QuePointer(MsgHdr, RetPtr)

MSGHDR *MsgHdr;
XANY **RetPtr;
```

PARAMETERS

Name	Description
`MsgHdr`	A pointer to a message header. *MsgHdr* refers to a message whose text is recorded in the Message Text Pool.
`RetPtr`	A pointer to the pointer variable that is returned with the text pointer; or `NULL` if no return value is desired.

RETURNS

Value	Description
RC >= 0	QuePointer successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QuePointer() obtains a pointer to the first byte (offset 0L) of the text associated with the message header it is passed. The pointer can then be used for directly accessing the message's text
Using QuePointer() to examine the contents of a message's text, without accessing its entire text and removing it from the message text pool, can be a very useful method for determining the importance of a message to a program's processing.
Manipulating a message's text space should generally be avoided. If it is necessary, great care should be employed to prevent against overstepping the boundary of the message's text space. Recall that the size of a message is stored as part of the message header (i.e., *MsgHdr->Size*).
QuePointer() will return with a valid pointer to the message's text if the instance involved is local to the calling program (on the same physical node). Requests for a pointer to a message's text regarding a network instance that is not local, return a `QUE_ER_NOTLOCAL` error code.

ERRORS

Code	Description
QUE_ER_BADBUFFER	*Ptr* is NULL.
QUE_ER_BADTEXT	The text pointer in *MsgHdr* is invalid.
QUE_ER_NOTLOCAL	Instance is not local.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).

INTERACTIVE COMMAND

SYNTAX

quepointer *MsgHdr*

ARGUMENTS

MsgHdr A one letter message header variable.

EXAMPLES

```
xipc> queget a ea 0 wait
      Qid = 0, Seq# = 1234, Prio = 1000, HdrStatus = REMOVED
xipc> quepointer a
      Pointer = 0000A03C
```

2.2.22 QuePurge() - PURGE A QUEUE

NAME

`QuePurge()` - Purge a Queue

SYNTAX

```
#include "xipc.h"


XINT
QuePurge(Qid)

XINT Qid;
```

PARAMETERS

Name	Description
Qid	The Queue ID of the queue to be Purged. *Qid* was obtained by the user via QueCreate() or QueAccess() function calls.

RETURNS

Value	Description
RC >= 0	Purge successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QuePurge() deletes all the messages from the Queue identified by *Qid* regardless of whether other users are waiting to send or receive messages via the queue. Users blocked on QueSys calls involving queue *Qid* (such as QueSend(), QueReceive(), QuePut() or QueGet()) are interrupted and returned an error code of `QUE_ER_PURGED` indicating the queue contents has been purged.
Note that since the queue is not deleted, queue statistics remain intact.

ERRORS

Code	Description
QUE_ER_BADQID	No queue with *Qid*.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

quepurge *Qid*

ARGUMENTS

Qid Queue Id.

EXAMPLES

```
xipc> quepurge 2
      RetCode = 0
```

2.2.23 QuePut() - PUT A MESSAGE HEADER ON A QUEUE

NAME

QuePut() - Put a Message Header on a Queue

SYNTAX

```
#include "xipc.h"


XINT
QuePut(MsgHdr, QueSelectCode, QidList, Priority, RetQid, Options)

MSGHDR *MsgHdr;
XINT QueSelectCode;
QIDLIST QidList;
XINT Priority;
XINT *RetQid;
... Options;
```

PARAMETERS

Name	Description
MsgHdr	A pointer to a message header.
QueSelectCode	A code indicating the selection criteria to be used in determining the target queue of the QuePut() operation. The selected queue is one of the Qid's *QidList*. The possible values for *QueSelectCode* are listed in section 2.3.2.
QidList	A list of Qids for consideration as the target queue of the QuePut() operation. A QIDLIST is constructed using QueList() or QueListBuild() and is updated using QueListAdd(). A pointer to a QIDLIST (type PQIDLIST) may be passed as well.
Priority	The priority to be assigned to the dispatched message.
RetQid	A pointer to a variable that gets assigned by QuePut() upon its return; or `NULL` if no return value is desired. Successful QuePut() operations (RC >= 0) return with **RetQid* equal to the Qid of the selected target queue (from within QidList) that received the sent message. Cancelled QuePut() operations having RC = `QUE_ER_DESTROYED` or `QUE_ER_PURGED`, return with **RetQid* equal to the destroyed or purged Qid. Failed calls with `RC = QUE_ER_BADQID` return with **RetQid* equal to the invalid Qid. **RetQid* is otherwise undefined.
Options	*Options* must be one of the following:

a) valid *BlockOpt* option. See Appendix A, Using Blocking X•IPC Functions, for a description of *BlockOpt*;

OR

b) one of the QUE_REPLACE_*XX* options (identified in the Description section which follows). Specifying QUE_REPLACE_*XX* causes the QuePut operation to succeed without blocking and *without* the need for spooling. QuePut(...,QUE_REPLACE_*XX*) succeeds by deleting one or more existing messages from the queue as necessary, if room needs to be made for the new message. See the Description below for a discussion of which messages are deleted from the queue in making room for the new message;

OR

c) QUE_REPLICATE, which sends message copies to those processes that are waiting for this message at the time of the QuePut operation. (See the Description below.)

RETURNS

Value	Description
RC >= 0	Put successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QuePut() attempts to put the message header *MsgHdr* onto one of the queues listed in *QidList*, based on the value of *QueSelectCode*.
It is acceptable to specify NULL for the *RetQid* argument value; it is not necessary to declare and specify return variables for acquiring a return value that is not desired.
QuePut() is given the potential to block or complete asynchronously by setting *BlockOpt* appropriately. The operation will block or complete asynchronously if all of the queues listed in *QidList* are currently filled to their message or byte capacities. The QuePut() operation will complete when one of queues in *QidList* releases a message, also freeing up the required bytes, usually through another user's actions. Specifically:

- Another user calling QueGet() or QueReceive() to take a message off of one of the involved queues.

Alternatively, the QUE_REPLACE_*XX* option can be used to specify that the operation should succeed without blocking and without the need for spooling. Specifying QuePut(...,QUE_REPLACE_*XX*) causes the enqueue operation to always succeed by deleting one or more existing messages from the queue, if room needs to be made for the new message. The deleted message may come from one of the four open ends of the message queue: *Earliest Arrived, Latest Arrived, Highest Priority , Lowest Priority*. Specifying the "end" from where messages are to be deleted is accomplished by specifying one of four forms of the QUE_REPLACE_*XX* option: QUE_REPLACE_HP, QUE_REPLACE_LP, QUE_REPLACE_EA, QUE_REPLACE_LA. This feature simplifies the construction of applications that have no assigned "remover of old messages," (e.g., an application that replicates messages to be sent to multiple users over a single queue). QUE_REPLACE_*XX* is a full-fledged completion option; it is not ORed with any of the six regular *X•IPC* completion options.
Alternatively, the QUE_REPLICATE option may be specified. This option provides a method for putting replicated message copies for zero or more users waiting on a message queue for that particular kind of message. (Note that no special coding is required by the consumer processes.) In this case, the messages are never actually placed on the queue. Messages are sent to only those processes that are waiting *at the time* of the QuePut() operation. All users waiting for the message are given a copy of the message header. When QuePut() replicates a header, copying to *n* users, the text-block reference

count is incremented by *n-1*. In contrast, when QuePut() moves a header onto a queue, the count is left *unchanged*.
As with the QUE_REPLACE_*XX* option, specifying QUE_REPLICATE causes the QuePut() call not to block. Similarly, QUE_REPLICATE is a full fledged completion option; it is not ORed with any of the six regular *X•IPC* completion options.
See Appendix A, Using Blocking *X•IPC* Functions, for a description of how to use the blocking options.
The MSGHDR data structure is defined as follows:

```
typedef struct _MSGHDR
{
    XINT GetQid;                    /* Last Qid msg was on */
    XINT HdrStatus;                     /* Rmvd or Not Rmvd, etc */
    XINT Priority;                      /* Message's priority */
    XINT SeqNum;                    /* Msg sequence # within queue */
    XINT TimeVal;                       /* Msg sequence number within QueSys */
    XINT Size;                          /* Numb. of bytes in msg */
    XINT TextOffset;                /* Offset of msg's text in text-pool */
    XINT Uid;                           /* The User-Id of user that sent msg */
    CHAR Data[MSGHDR_DATASIZE];         /* User data field */
}
MSGHDR;
```

ERRORS

Code	Description
QUE_ER_ASYNC	Operation is being performed asynchronously.
QUE_ER_ASYNCABORT	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
QUE_ER_BADBLOCKOPT	Invalid *BlockOpt*.
QUE_ER_BADOPTION	Invalid *Options* parameter.
QUE_ER_BADPRIORITY	Invalid *Priority* parameter.
QUE_ER_BADQID	Bad Qid in QidList (= **RetQid*).
QUE_ER_BADQIDLIST	Invalid *QidList* parameter.
QUE_ER_BADQUESELECTCODE	Invalid QueSelectCode parameter.
QUE_ER_CAPACITY_ASYNC_USER	QueSys async user table full.
QUE_ER_CAPACITY_HEADER	QueSys header table full.
QUE_ER_CAPACITY_NODE	QueSys node table full.
QUE_ER_DESTROYED	Another user destroyed a queue that the blocked QuePut() call was waiting on (i.e. in its QidList). The blocked QuePut() operation was cancelled. No message was put on any queue.
QUE_ER_INTERRUPT	Operation was interrupted.
QUE_ER_ISFROZEN	A *BlockOpt* of QUE_WAIT or QUE_TIMEOUT() was specified after the instance was frozen by the calling user.
QUE_ER_MSGHDRNOTREMOVED	*MsgHdr* references a message header that is still on a queue.
QUE_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.

QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
QUE_ER_NOWAIT	*BlockOpt* of QUE_NOWAIT was specified and request was not immediately satisfied.
QUE_ER_PURGED	Another user purged a queue that the blocked QuePut() call was waiting on (i.e. in its QidList). The blocked QuePut() operation was cancelled. No message was put on any queue.
QUE_ER_TIMEOUT	The blocked QuePut() operation timed out.
QUE_ER_TOOBIG	The size of the message exceeds the byte capacity of one of the listed Qids (= **RetQid*).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

queput *MsgHdr QueSelectCode QidList Priority BlockingOpt*

ARGUMENTS

MsgHdr A one letter message header variable.

QueSelectCode A message dispatch queue select code. (This is presented in the QueSys Functions and Macros chapter, under Macros.) The prefix "QUE_Q_" of the queue select code should be omitted, e.g., instead of QUE_Q_SHQ, use **shq**.

QidList A list of Queue Ids

Priority The priority to be assigned to the message.

BlockingOpt Either one of the Blocking Options discussed in the xipc command (Interactive Command Processor) section at the beginning of this Manual, or one of the special blocking options (REPLICATE, REPLACE_EA, REPLACE_LA, REPLACE_HP or REPLACE_LP).

EXAMPLES

```
xipc> quewrite a "Mary had a little lamb" wait
      RetCode = 0
xipc> queput a shq 0,1 100 wait
      RetCode = 0  Qid = 1
```

2.2.24 QueRead() - READ MESSAGE TEXT FROM MESSAGE TEXT POOL

NAME

QueRead() - Read Message Text From Message Text Pool

SYNTAX

```
#include "xipc.h"


XINT
QueRead(MsgHdr, MsgBuf, MsgLength)

MSGHDR *MsgHdr;
XANY *MsgBuf;
XINT MsgLength;
```

PARAMETERS

Name	Description
MsgHdr	A pointer to a message header. *MsgHdr* contains information relating to a message that has been removed from a queue, but whose text is still in the message text pool.
MsgBuf	A pointer to the buffer that is to receive the text of the message referred to by *MsgHdr*.
MsgLength	An integer specifying the maximum number of bytes to be copied from the message's text area into *MsgBuf*. QUE_TRUNCATE(*MsgLength*) must be specified if text truncation is desired for messages exceeding *MsgLength* bytes in size. *MsgLength* must be greater than 0.

RETURNS

Value	Description
RC > 0	Read successful. RC is the number of bytes copied into *MsgBuf*.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueRead() reads the message text referred to by *MsgHdr* into the buffer pointed at by *MsgBuf*. (The message header can have been retrieved using the QUE_NOREMOVE option or a QueCopy() call.) QueRead() releases the text pool area holding the message text. As such, QueRead() is usually called at a message's final destination.
QueRead() decrements the text pool count by one; if the count equals zero, then it will release the text block.

If the size of the message's text is less than or equal to *MsgLength* bytes, the message is copied in its entirety into *MsgBuf*. If the size is larger, then:

- If QUE_TRUNCATE(MsgLength) is specified, the first MsgLength bytes of the text are copied into MsgBuf. The remaining bytes are truncated.
- Otherwise, QueRead() fails, returning RC = QUE_ER_TOOBIG.

ERRORS

Code	Description
QUE_ER_BADBUFFER	*MsgBuf* is NULL.
QUE_ER_BADLENGTH	Invalid *MsgLength* parameter.
QUE_ER_BADTEXT	The text pointer in *MsgHdr* is invalid.
QUE_ER_MSGHDRNOTREMOVED	*MsgHdr* references a message header that has not been dequeued.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
QUE_ER_TOOBIG	The size of the message's text exceeds *MsgLength* and QUE_TRUNCATE was not specified.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.
XIPCNET_ER_TOOBIG	Text exceeds instance's size limit.

INTERACTIVE COMMAND

SYNTAX

queread *MsgHdr*

ARGUMENTS

MsgHdr A one letter message header variable.

EXAMPLES

```
xipc> queget a ea 1 wait
      RetCode = 0, Qid = 1, Seq# = 1211, Prio = 100, HdrStatus = REMOVED
xipc> queread a
      Text = "Mary had a little lamb"
```

2.2.25 QueReceive() - RECEIVE AND READ A MESSAGE FROM A QUEUE

NAME

QueReceive() - Receive and Read a Message From a Queue

SYNTAX

```
#include "xipc.h"


XINT
QueReceive(QueSelectCode, QidList, MsgBuf, MsgLength,
           RetVal, RetQid, Options)

XINT QueSelectCode;
QIDLIST QidList;
XANY *MsgBuf;
XINT MsgLength;
XINT *RetVal;
XINT *RetQid;
... Options;
```

PARAMETERS

Name	Description
QueSelectCode	A code indicating the selection criteria to be used in determining the received message of the QueReceive() operation. The selected message is taken from one of the Qids in *QidList*. The possible values for *QueSelectCode* are listed in section 2.3.2.
QidList	A list of Qids, possibly specified within Message Select Code macros, to be used in specifying candidate messages for consideration by the QueReceive() operation. QueReceive() selects one of the candidate messages based on the value of *QueSelectCode*. A QIDLIST is constructed using QueList() or QueListBuild() and is updated using QueListAdd(). A pointer to a QIDLIST (type PQIDLIST) may be passed as well.
MsgBuf	A pointer to the message buffer to receive the message text.
MsgLength	An integer specifying the maximum number of bytes to be copied from the retrieved message into *MsgBuf*. `QUE_TRUNCATE(MsgLength)` must be specified if text truncation is desired for messages exceeding this size. *MsgLength* must be greater than 0.
RetVal	A pointer to a 32-bit integer variable that gets assigned with either the received message's priority or its sequence number. It can be `NULL` if no return value is desired.
RetQid	A pointer to a variable that gets assigned by QueReceive() upon its return; or `NULL` if no return value is desired. Successful QueReceive() operations (RC >= 0) return with **RetQid* equal to the Qid of the selected source queue (from

within QidList) that provided the received message. Cancelled QueReceive() operations having RC = QUE_ER_DESTROYED or QUE_ER_PURGED, return with **RetQid* equal to the destroyed or purged Qid. Failed calls with RC = QUE_ER_BADQID return with **RetQid* equal to the invalid Qid. **RetQid* is otherwise undefined.

Options [*OptionFlag* | …] *BlockOpt*

OptionFlags are specified by ORing them to the left of the *BlockOpt*. An example will follow below. The possible *OptionFlags* are:

- QUE_NOREMOVE - When specified, the accessed message is *not* dequeued. Rather, a fully functional copy of the message is retrieved. The actual message remains on the queue.

The following two flags are mutually exclusive; QUE_RETPRIO is the default:

- QUE_RETPRIO - Specifies retrieval of the Priority of the retrieved message.
- QUE_RETSEQ - Specifies retrieval of the Sequence Number of the retrieved message.
- *BlockOpt* specifies the blocking option. See the Using Blocking *X•IPC* Functions Appendix for a description of *BlockOpt*.

Example: QueReceive(…, QUE_RETSEQ | QUE_NOREMOVE | QUE_WAIT)

RETURNS

Value	Description
RC >= 0	Receive successful. RC is the number of bytes copied into *MsgBuf*.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueReceive() attempts to receive a QueSys message from one of the Qids in *QidList*. The determination of which message is received is based on each Qid's Message Select Code as listed in *QidList*, in conjunction with the value of *QueSelectCode*. QueReceive() first attempts to access the chosen message header, and remove it from its queue. It then reads the text of the message from the message text pool into the buffer pointed at by *MsgBuf*. *RetVal* is assigned with the retrieved message's priority or sequence number. The message's text pool area is then released.
If the size of the selected message is less than or equal to *MsgLength* bytes, the message is copied in its entirety into *MsgBuf*. If the message size is larger, then:

- If QUE_TRUNCATE(*MsgLength*) is specified, the first *MsgLength* bytes of the message are copied into *MsgBuf*. The remaining bytes are truncated and lost.
- Otherwise, QueReceive() fails, returning RC = QUE_ER_TOOBIG and the message remains in the queue.

It is acceptable to have a null *RetQid* or *RetVal* argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired. In addition, QueReceive() can specify whether the returned *RetVal* should be populated with the priority of the retrieved message or the

sequence number of the retrieved message. This is accomplished by using one of the two optional flags, QUE_RETPRIO or QUE_RETSEQ, which determine which value is returned with the message. The default value is QUE_RETPRIO.
Calling QueReceive() with the QUE_NOREMOVE option flag specified returns a fully functional copy of the message. The actual message remains on the queue.
QueReceive() is given the potential to block or complete asynchronously by setting *BlockOpt* appropriately. The operation will block or complete asynchronously if either all the listed queues are empty, or contain messages not matching their respective Message Select Codes. A QueReceive() operation will complete when the cause of it not completing is removed, usually by another user's actions. Specifically:

- Another user calling QueSend() or QuePut() to place a message on one of the involved queues.

See Appendix A, Using Blocking X•IPC Functions, for a description of how to use the blocking options.

ERRORS

Code	Description
QUE_ER_ASYNC	Operation is being performed asynchronously.
QUE_ER_ASYNCABORT	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
QUE_ER_BADBLOCKOPT	Invalid *BlockOpt*.
QUE_ER_BADBUFFER	*MsgBuf* is NULL.
QUE_ER_BADLENGTH	Invalid *MsgLength* parameter.
QUE_ER_BADMSGSELECTCODE	Invalid MsgSelectCode within *QidList*.
QUE_ER_BADOPTION	Invalid *Options* parameter.
QUE_ER_BADQID	Bad Qid in QidList (= **RetQid)*.
QUE_ER_BADQIDLIST	Invalid QidList parameter.
QUE_ER_BADQUESELECTCODE	Invalid QueSelectCode parameter.
QUE_ER_CAPACITY_ASYNC_USER	QueSys async user table full.
QUE_ER_CAPACITY_NODE	QueSys node table full.
QUE_ER_DESTROYED	Another user destroyed a queue that the blocked QueReceive() call was waiting on. The blocked QueReceive() operation was cancelled. No message was received.
QUE_ER_INTERRUPT	Operation was interrupted.
QUE_ER_ISFROZEN	A *BlockOpt* of QUE_WAIT or QUE_TIMEOUT() was specified after the instance was frozen by the calling user.
QUE_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
QUE_ER_NOWAIT	*BlockOpt* of QUE_NOWAIT was specified and request was not immediately satisfied.
QUE_ER_PURGED	Another user purged a queue that the blocked QueReceive() call was waiting on. The blocked QueReceive() operation was cancelled. No message was received.

QUE_ER_TIMEOUT	The blocked QueReceive() operation timed out.
QUE_ER_TOOBIG	The size of the message exceeds *MsgLength* and QUE_TRUNCATE was not specified.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.
XIPCNET_ER_TOOBIG	Message text exceeds instance's size limit.

INTERACTIVE COMMAND

SYNTAX

quereceive *QueSelectCode QidList*
[noremove,][**retprio**,|**retseq**,] *BlockingOpt*

ARGUMENTS

QueSelectCode A message retrieval queue select code. (This is presented in the QueSys Functions and Macros chapter, under Macros.) The prefix "QUE_Q_" of the queue select code should be omitted, e.g., instead of QUE_Q_EA, use **ea**.

QidList A list of queue Ids, possibly specified with message select codes. (This is presented in the QueSys Functions and Macros chapter, under Macros.) The prefix "QUE_M_" of the message select code should be omitted, e.g., instead of QUE_M_PRGT(2,100), use **prgt(2,100)**.

BlockingOpt See the Blocking Options discussion in the xipc command (Interactive Command Processor) section at the beginning of this Manual.

EXAMPLES

```
xipc> quereceive ea 0 timeout(10)
      Qid = 0, Priority = 120, Length = 22
      Text = "Mary had a little lamb"

xipc> quereceive hpq prgt(0,1000),prgt(1,900) wait
      Qid = 1, Priority = 950, Length = 13
      Text = "High Priority"

xipc> quereceive ea 0 retseq,wait
      Qid = 0, Sequence = 1, Length = 22
      Text = "Mary had a little lamb"

xipc> quereceive ea 0 retprio,wait
      Qid = 0, Priority = 120, Length = 22
      Text = "Mary had a little lamb"

xipc> quereceive ea 0 noremove,wait
      Qid = 0, Priority = 120, Length = 22
      Text = "Mary had a little lamb"
```

```
xipc> quereceive ea 0 noremove,retseq,nowait
      Qid = 0, Sequence = 12, Length = 23
      Text = "System uptime: 12:33:02"
```

2.2.26 QueRemove() - REMOVE MESSAGE HEADER FROM A QUEUE

NAME

QueRemove() - Remove Message Header from a Queue.

SYNTAX

```
#include "xipc.h"


XINT
QueRemove(MsgHdr)

MSGHDR *MsgHdr;
```

PARAMETERS

Name	Description
MsgHdr	Pointer to a message header variable that contains a fully functional cop of a message header still residing on a queue.

RETURNS

Value	Description
RC > 0	Remove successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueRemove() removes a fully functional copy of the message header identified by *MsgHdr* from the message queue where it is located. The message header identified by *MsgHdr* must have been accessed previously by a call to QueGet() specifying the QUE_NOREMOVE option (placed to the left of the blocking option) or by a call to QueBrowse(). The common factor being that *MsgHdr* references a message header that has not been dequeued.
A message header that is removed from a queue via QueRemove() may be placed onto another queue via QuePut(), or can have its text read from the message text pool via a call to QueRead().

ERRORS

Code	Description
QUE_ER_BADTEXT	The text pointer in *MsgHdr* is invalid.
QUE_ER_MSGHDRREMOVED	*MsgHdr* has already been dequeued.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

queremove *MsgHdr*

ARGUMENTS

MsgHdr A one letter message header variable.

EXAMPLES

```
xipc> queget a ea 0 noremove,wait
      RetCode = 0, Qid = 1, Seq# = 1011, Prio = 100, HdrStatus = NOT-REMOVED
xipc> quebrowse a time+
      RetCode = 0, Qid = 1, Seq# = 1211, Prio = 100, HdrStatus = NOT-REMOVED
xipc> queremove a
      RetCode = 0
```

2.2.27 QueSend() - WRITE AND SEND A MESSAGE TO A QUEUE

NAME

QueSend() - Write and Send a Message To a Queue

SYNTAX

```
#include "xipc.h"


XINT
QueSend(QueSelectCode, QidList, MsgBuf, MsgLength, Priority,
         RetQid, Options)

XINT QueSelectCode;
QIDLIST QidList;
XANY *MsgBuf;
XINT MsgLength;
XINT Priority;
XINT *RetQid;
... Options;
```

PARAMETERS

Name	Description
QueSelectCode	A code indicating the selection criteria to be used in determining the target queue of the QueSend() operation. The selected queue is one of the Qids in *QidList*. The possible values for *QueSelectCode* are listed in section 2.3.2.
QidList	A list of Qids for consideration as the target queue of the QueSend() operation. A QIDLIST is constructed using QueList() or QueListBuild() and is updated using QueListAdd(). A pointer to a QIDLIST (type PQIDLIST) may be passed as well.
MsgBuf	A pointer to the message text to be sent.
MsgLength	The size (in bytes) of the message in *MsgBuf*. Its value must be greater than 0.
Priority	A positive 32-bit integer to be designated as the message's priority.
RetQid	A pointer to a variable that gets assigned by QueSend() upon its return; or `NULL` if no return value is desired. Successful QueSend() operations (RC >= 0) return with **RetQid* equal to the Qid of the selected target queue (from within QidList) that received the sent message. Cancelled QueSend() operations having RC = `QUE_ER_DESTROYED` or `QUE_ER_PURGED`, return with **RetQid* equal to the destroyed or purged Qid. Failed calls with `RC = QUE_ER_BADQID` return with **RetQid* equal to the invalid Qid. **RetQid* is otherwise undefined.
Options	*Options* must be one of the following:

a) valid BlockOpt option. See Appendix A, Using Blocking *X•IPC* Functions, for a description of BlockOpt;
OR

b) one of the QUE_REPLACE_*XX* options, (identified in the Description section which follows). Specifying QUE_REPLACE_*XX* causes the QueSend() operation to succeed without blocking and without the need for spooling. QueSend(...,QUE_REPLACE_*XX*) succeeds by deleting one or more existing messages from the queue as necessary, if room needs to be made for the new message. See the Description below for a discussion of which messages are deleted from the queue in making room for the new message;

OR

c) QUE_REPLICATE, which sends message copies to those processes that are waiting for this message at the time of the QueSend() operation. (See the Description below.)

RETURNS

Value	Description
RC >= 0	Send successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueSend() attempts to write the message text in *MsgBuf* into the QueSys message text pool and then put a header referring to it onto one of the queues listed in *QidList*. The selection of a target queue is based on the value of *QueSelectCode*. QueSend() first attempts to write the message's text to the QueSys message text pool, creating for it a message header (i.e., as per QueWrite()). It then attempts to put the created message header onto one of the queues in *QidList* based on the value of *QueSelectCode* (i.e., as per QuePut()).
It is acceptable to specify NULL as the *RetQid* argument value; it is not necessary to declare and specify a return variable for acquiring a return value that is not desired.
QueSend() is given the potential to block or complete asynchronously by setting *BlockOpt* appropriately. The operation will block or complete asynchronously in the following cases:

- The message text pool currently lacks the capacity for a message text of size *MsgLength*.
- All the queues listed in *QidList* are currently filled to their message or byte capacities.

A QueSend() operation completes when the cause of it not completing is removed, usually by another user's actions. Specifically:

- Another user calling QueRead() to remove a message's text from the message text pool.
- Another user calling QueGet() to retrieve a message from one of the involved queues.
- Another user calling QueReceive() to accomplish both effects.

Alternatively, the QUE_REPLACE_*XX* option can be used to specify that the operation should succeed *without* blocking and without the need for spooling. Specifying QueSend(...,QUE_REPLACE_*XX*) causes the enqueue operation to always succeed by deleting one or more existing messages from the queue, if room needs to be made for the new message. The deleted message may come from one of the four open ends of the message queue: *Earliest Arrived, Latest Arrived, Highest Priority , Lowest Priority*. Specifying the "end" from where messages are to be deleted is accomplished by specifying one of four forms of the QUE_REPLACE_*XX* option:

QUE_REPLACE_HP, QUE_REPLACE_LP, QUE_REPLACE_EA, QUE_REPLACE_LA.
This feature simplifies the construction of applications that have no assigned "remover of old messages," (e.g., an application that replicates messages to be sent to multiple users). QUE_REPLACE_*XX* is a full-fledged completion option. It is not ORed with any of the six regular X•IPC completion options. If a QueSend() is involved and the write to the text-pool is not possible because of it being full, the QueSend() will return an error code indicating the inability to perform the operation, i.e., QUE_ER_TEXTFULL.
Alternatively, the QUE_REPLICATE option may be specified. This option provides a method for sending replicated message copies to zero or more users waiting on a message queue for that particular kind of message. (Note that no special coding is required by the consumer processes.) In this case, the messages are never actually placed on the queue. Messages are sent to only those processes that are waiting *at the time* of the QueSend() operation. All users waiting for the message are given a copy of the message header. When QueSend() replicates a message, copying to *n* users, the text-block reference count is incremented by *n-1*. In contrast, when QueSend() places a message on a queue, the count is left *unchanged*.
See Appendix A, Using Blocking X•IPC Functions, for a description of how to use the blocking options.

ERRORS

Code	Description
QUE_ER_ASYNC	Operation is being performed asynchronously.
QUE_ER_ASYNCABORT	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
QUE_ER_BADBLOCKOPT	Invalid *BlockOpt*.
QUE_ER_BADBUFFER	*MsgBuf* is NULL.
QUE_ER_BADFILENAME	Invalid *SpoolFileName* specified.
QUE_ER_BADLENGTH	Invalid *MsgLength* parameter.
QUE_ER_BADOPTION	Invalid *Options* parameter.
QUE_ER_BADPRIORITY	Invalid *Priority* parameter.
QUE_ER_BADQID	Bad Qid in QidList (= **RetQid*).
QUE_ER_BADQIDLIST	Invalid QidList parameter.
QUE_ER_BADQUESELECTCODE	Invalid QueSelectCode parameter.
QUE_ER_CAPACITY_ASYNC_USER	QueSys async user table full.
QUE_ER_CAPACITY_HEADER	QueSys header table full.
QUE_ER_CAPACITY_NODE	QueSys node table full.
QUE_ER_DESTROYED	Another user destroyed a queue that the blocked QueSend() call was waiting on. The blocked QueSend() operation was cancelled. No message was sent.
QUE_ER_INTERRUPT	Operation was interrupted.
QUE_ER_ISFROZEN	A *BlockOpt* of QUE_WAIT or QUE_TIMEOUT() was specified after the instance was frozen by the calling user.
QUE_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
QUE_ER_NOWAIT	*BlockOpt* of QUE_NOWAIT was specified and request was

	not immediately satisfied.
QUE_ER_PURGED	Another user purged a queue that the blocked QueSend() call was waiting on. The blocked QueSend() operation was cancelled. No message was sent.
QUE_ER_TIMEOUT	The blocked QueSend() operation timed out.
QUE_ER_TOOBIG	The size of the message exceeds the byte capacity of one of the listed Qids (= **RetQid*).
QUE_ER_TEXTFULL	Text space is not available when QUE_REPLICATE or QUE_REPLACE_*XX* is specified, causing call to fail.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.
XIPCNET_ER_TOOBIG	Message text exceeds instance's size limit.

INTERACTIVE COMMAND

SYNTAX

quesend *QueSelectCode QidList Priority MessageText BlockingOpt*

ARGUMENTS

QueSelectCode	A message dispatch queue select code. (This is presented in the QueSys Functions and Macros chapter, under Macros.) The prefix "QUE_Q_" of the queue select code should be omitted, e.g., instead of QUE_Q_SHQ, use **shq**.
QidList	A list of queue Ids
Priority	The priority to be assigned to the message.
MessageText	The text of the message enclosed in double quotes.
BlockingOpt	Either one of the Blocking Options discussed in the xipc command (Interactive Command Processor) section at the beginning of this Manual, or one of the special blocking options (REPLICATE, REPLACE_EA, REPLACE_LA, REPLACE_HP or REPLACE_LP).

EXAMPLES

```
xipc> quesend shq 0,1 100 "Mary had a little lamb" wait
      RetCode = 0  Qid = 0
```

2.2.28 QueSendReceive() - PERFORM GENERIC REQUEST/RESPONSE

NAME

QueSendReceive() - Perform Generic Request/Response

SYNTAX

```
#include "xipc.h"

QueSendReceive (SendArgs, SendOptions, RecvArgs, RecvOptions)

QUE_SEND_ARGS       *SendArgs;
. . .               SendOptions;
QUE_RECEIVE_ARGS       *RecvArgs;
. . .               RecvOptions;
```

PARAMETERS

Name	Description
SendArgs	A pointer to a variable of type QUE_SEND_ARGS (structure is defined below in the Description), containing the arguments for the "send" portion of the QueSendReceive() operation.
SendOptions	See the *Options* parameter description under the QueSend() API. See also Appendix A, Using Blocking Options.
RecvArgs	A pointer to a variable of type QUE_RECEIVE_ARGS (structure is defined below in the Description), containing the arguments for the "receive" portion of the QueSendReceive() operation.
RecvOptions	See the *Options* parameter description under the QueReceive() API. See also Appendix A, Using Blocking Options.

RETURNS

Value	Description
RC >= 0	SendReceive request successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueSendReceive () performs analogously to the RPC request/response paradigm by performing a QueSend() operation immediately followed by a QueReceive() operation with a *single* network operation.

The usage of queues within QueSendReceive() is highly flexible. For example, a client "inquiry" message may be sent to a server via one queue and a "response" message drawn from a second queue. Similarly, by specifying the receive operation to execute asynchronously, one can cause the inquiry-

response interaction to complete in the background (e.g., with callback functions invoked at the client whenever a "response" message arrives).

It is important to note that unlike traditional RPC mechanisms, the QueSendReceive() form of inquiry-response functionality provides explicit message queuing elasticity for handling high-volume traffic scenarios. This is critical when preparing a system that must scale well through a range of deployment settings.

The data structure follows:

```
typedef _QUE_SEND_ARGS
{
    XINT        QueSelectCode;
    PQIDLIST    QidList;
    XANY            *MsgBuf;
    XINT        MsgLength;
    XINT        Priority;
    XINT        *QidPtr;
}
QUE_SEND_ARGS;

typedef _QUE_RECEIVE_ARGS
{
    XINT        QueSelectCode;
    PQIDLIST    QidList;
    XANY            *MsgBuf;
    XINT        MsgLength;
    XINT        *PrioPtr;
    XINT        *QidPtr;
}
QUE_RECEIVE_ARGS;
```

ERRORS

Code	**Description**
QUE_ER_ASYNC	Operation is being performed asynchronously.
QUE_ER_ASYNCABORT	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
QUE_ER_BADBLOCKOPT	Invalid *BlockOpt*.
QUE_ER_BADBUFFER	*MsgBuf* is NULL.
QUE_ER_BADLENGTH	Invalid *MsgLength* parameter.
QUE_ER_BADMSGSELECTCODE	Invalid MsgSelectCode within *QidList*.
QUE_ER_BADOPTION	Invalid *Options* parameter.
QUE_ER_BADPRIORITY	Invalid *Priority* parameter.
QUE_ER_BADQID	Bad Qid in QidList (= **QidPtr*).
QUE_ER_BADQIDLIST	Invalid QidList parameter.
QUE_ER_BADQUESELECTCODE	Invalid QueSelectCode parameter.
QUE_ER_CAPACITY_ASYNC_USER	QueSys async user table full.

QUE_ER_CAPACITY_HEADER	QueSys header table full.
QUE_ER_CAPACITY_NODE	QueSys node table full.
QUE_ER_DESTROYED	Another user destroyed a queue that the blocked QueSend() call was waiting on. The blocked QueSend() operation was cancelled. No message was sent.
QUE_ER_INTERRUPT	Operation was interrupted.
QUE_ER_ISFROZEN	A *BlockOpt* of QUE_WAIT or QUE_TIMEOUT() was specified after the instance was frozen by the calling user.
QUE_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
QUE_ER_NOWAIT	*BlockOpt* of QUE_NOWAIT was specified and request was not immediately satisfied.
QUE_ER_PURGED	Another user purged a queue that the blocked QueSend() call was waiting on. The blocked QueSend() operation was cancelled. No message was sent.
QUE_ER_TIMEOUT	The blocked QueSend() operation timed out.
QUE_ER_TOOBIG	The size of the message exceeds the byte capacity of one of the listed Qids (= **QidPtr*).
QUE_ER_TEXTFULL	Text space is not available when QUE_REPLICATE or QUE_REPLACE_*XXX* is specified, causing call to fail.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.
XIPCNET_ER_TOOBIG	Message text exceeds instance's size limit.

INTERACTIVE COMMAND

SYNTAX

```
quesendreceive  SendQueSelectCode SendQidList Priority
MessageText SendBlockOpt              RecvQueSelectCode
RecvQidList [noremove,] [retseq,|retprio,]
     RecvBlockOpt
```

ARGUMENTS

SendQueSelectCode	A message dispatch queue select code. (This is presented in the QueSys Functions and Macros chapter, under Macros.) The prefix "QUE_Q_" of the queue select code should be omitted, e.g., instead of QUE_Q_SHQ, use **shq**.
SendQidList	A list of queue Ids
Priority	The priority to be assigned to the message.
MessageText	The text of the message enclosed in double quotes.
SendBlockOpt	One of **wait**, **nowait** or **timeout(**Seconds**)**.
RecvQueSelect	A message retrieval queue select code. (This is presented in the QueSys Functions and Macros chapter, under Macros.) The prefix "QUE_Q_" of the queue select code should be omitted, e.g., instead of QUE_Q_EA, use **ea**.
RecvQidList	A list of queue Ids, possibly specified with message select codes. (This is presented in the QueSys Functions and Macros chapter, under Macros.) The prefix "QUE_M_" of the message select code should be omitted, e.g., instead of QUE_M_PRGT(2,100), use **prgt(2,100)**.
RecvBlockOpt	See the Blocking Options discussion in the xipc command (Interactive Command Processor) section at the beginning of this Manual.

EXAMPLES

```
xipc> quesend any 0 100 "send to qid 0" nowait
      RetCode = 0, Qid = 0

xipc> quesendreceive any 1 99 "sent to qid 1" nowait any ea(0) nowait
      RetCode = 13, SendQid = 1, RecvQid = 0,
      RecvLength = 13, RecvPriority = 100.
      Received textg: "sent to qid 0"

xipc> quereceive any ea(1) nowait
      Qid = 1, Priority = 99, Length = 13
      Text = "sent to qid 1"
```

2.2.29 QueSpool() - START AND STOP SPOOLING FOR A QUEUE

NAME

QueSpool() - Start and Stop Spooling for a Message Queue

SYNTAX

```
#include "xipc.h"


XINT
QueSpool(Qid, SpoolFileName)

XINT Qid;
CHAR *SpoolFileName;
```

PARAMETERS

Name	Description
Qid	The Queue ID of the queue to start or stop spooling.
SpoolFileName	The name of a spool file where spooling is to occur; or QUE_SPOOL_OFF if spooling is being turned off. *SpoolFileName* length must not exceed QUE_LEN_PATHNAME.

RETURNS

Value	Description
RC >= 0	Spool request successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueSpool() is used to start or stop overflow message spooling for a given queue. If *SpoolFileName* is QUE_SPOOL_OFF, spooling for the queue is turned off.

Otherwise, message spooling for the queue is enabled. Overflow messages are spooled using a group of files having *SpoolFileName* as their base name. Refer to the X•IPC User Guide for a detailed description of the spooling mechanism.

A queue's spool is a virtual extension of the queue. Spooled messages, however, do not participate in message retrieval competition involving the queue until they are absorbed from the spool into the queue proper.

Turning spooling "on" or "off" has no effect on messages already present on the queue's spool. It only effects the treatment of future messages attempting to enter the queue when full. When spooling is off, the queue can potentially block a QuePut() or a QueSend() operation. When spooling is on, the queue is by definition never full. QuePut() and QueSend() operations involving the queue will thus never block.

When spooling is turned "on" for a queue that currently has messages spooled, the specified *SpoolFileName* is ignored and spooling continues using the spool file name holding the previously spooled messages.

Spooling within a network instance occurs in the file system of the machine upon which the instance was started. The *SpoolFileName* argument must therefore conform to the file naming conventions of that platform.
Specifying a *SpoolFileName* that is currently being used by *another* queue will cause unpredictable results.

ERRORS

Code	**Description**
QUE_ER_BADFILENAME	Invalid *SpoolFileName* parameter.
QUE_ER_BADQID	Bad *Qid*.
QUE_ER_CAPACITY_NODE	QueSys node table full.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

quespool *Qid SpoolFileName*

ARGUMENTS

Qid Queue Id.

SpoolFileName Name of spool file or **off**.

EXAMPLES

```
xipc> quespool 0 /tmp/splq0
      RetCode = 0

xipc> quespool 0 off
      RetCode = 0
```

2.2.30 QueTrigger() - DEFINE A QUESYS TRIGGER

NAME

`QueTrigger()` - Define a QueSys Trigger

SYNTAX

```
#include "xipc.h"


XINT
QueTrigger(Sid, TriggerSpec)

XINT Sid;
... TriggerSpec;
```

PARAMETERS

Name	Description
`Sid`	The Semaphore ID of the event semaphore to be set when the trigger event occurs. The *Sid* is obtained by SemCreate() or SemAccess() function calls.
`TriggerSpec`	Specification of the QueSys trigger event. The event is specified using a macro that defines the type of event and parameters such as *Qid* and threshold values. See the description below for a list of all trigger specifications.

RETURNS

Value	Description
RC >= 0	QueTrigger successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

A QueSys trigger is a logical link between a QueSys event and a SemSys event semaphore. The semaphore is automatically set when the QueSys event occurs.
A trigger is defined by:

- ☐ The ID of a SemSys event semaphore that will be set when the QueSys event occurs.
- ☐ A QueSys event specification.

The *Sid* of the event semaphore is obtained by calling SemSys functions SemCreate() or SemAccess().

The following table contains a list of all QueSys events that can be specified:

Trigger	Description
`QUE_T_BYTES_HIGH(`*`Qid, N`*`)`	Trigger event when number of bytes written to queue *Qid* becomes higher than *N* percent of queue bytes capacity.
`QUE_T_BYTES_LOW(`*`Qid,`*	Trigger event when number of bytes written to queue *Qid*

N)	becomes lower than *N* percent of queue bytes capacity.
QUE_T_MSGS_HIGH(*Qid, N*)	Trigger event when number of messages written to queue *Qid* becomes higher than *N* percent of queue messages capacity.
QUE_T_MSGS_LOW(*Qid, N*)	Trigger event when number of messages written to queue *Qid* becomes lower than *N* percent of queue messages capacity.
QUE_T_PUT(*Qid*)	Trigger event when a message is put onto queue *Qid*.
QUE_T_GET(*Qid*)	Trigger event when a message is removed from queue *Qid*.
QUE_T_PUT_PREQ(*Qid, P*)	Trigger event when a message of priority *P* is put onto queue *Qid*.
QUE_T_GET_PREQ(*Qid, P*)	Trigger event when a message of priority *P* is removed from queue *Qid*.
QUE_T_PUT_PRGT(*Qid, P*)	Trigger event when a message of priority greater then *P* is put onto queue *Qid*.
QUE_T_GET_PRGT(*Qid, P*)	Trigger event when a message of priority greater then *P* is removed from queue *Qid*.
QUE_T_PUT_PRLT(*Qid, P*)	Trigger event when a message of priority less then *P* is put onto queue *Qid*.
QUE_T_GET_PRLT(*Qid, P*)	Trigger event when a message of priority less then *P* is removed from queue *Qid*.
QUE_T_USER_PUT(*Qid, Uid*)	Trigger event when a message is put onto queue *Qid* by user *Uid*.
QUE_T_USER_GET(*Qid, Uid*)	Trigger event when a message is removed from queue *Qid* by user *Uid*.
QUE_T_POOL_HIGH(*N*)	Trigger event when the allocated size of the message text pool becomes higher than *N* percent of its capacity.
QUE_T_POOL_LOW(*N*)	Trigger event when the allocated size of the message text pool becomes lower than *N* percent of its capacity.
QUE_T_HEADER_HIGH(*N*)	Trigger event when the number of allocated message headers becomes higher that *N* percent of the capacity.
QUE_T_HEADER_LOW(*N*)	Trigger event when the number of allocated message headers becomes lower that *N* percent of the capacity.

ERRORS

Code	**Description**
QUE_ER_BADQID	*Qid* is not a valid queue ID.
QUE_ER_BADSID	*Sid* is not a valid semaphore ID.
QUE_ER_BADTRIGGERCODE	Bad trigger code.
QUE_ER_BADUID	*Uid* is not a valid user id.
QUE_ER_BADVAL	Illegal trigger parameter value.
QUE_ER_CAPACITY_NODE	QueSys node table full.
QUE_ER_DUPLICATE	Attempt to define a trigger that is already defined
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).

XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

quetrigger *Sid TriggerCode TriggerParms*

ARGUMENTS

Sid	Semaphore Id of the semaphore to be set when the QueSys trigger event occurs.
TriggerCode	Mnemonic code of the trigger. Note that the prefix "QUE_T_" of the trigger code should not be specified, e.g., QUE_T_BYTES_HIGH should be specified as **bytes_high**.
TriggerParms	Additional parameters depending on the type of trigger defined.

EXAMPLES

```
xipc> quecreate MsgQueue nolimit 20000
      Qid = 7
xipc> semcreate BytesHighSem clear
      Sid = 31
xipc> # Set Semaphore 31 when size of text in Queue 7 exceeds 80 percent
xipc> quetrigger 31 bytes_high 7 80
      RetCode = 0
```

2.2.31 QueUnfreeze() - UNFREEZE QUESYS

NAME

`QueUnfreeze()` - Unfreeze QueSys

SYNTAX

```
#include "xipc.h"


XINT
QueUnfreeze()
```

PARAMETERS

None.

RETURNS

Value	Description
RC >= 0	QueUnfreeze successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueUnfreeze() unfreezes QueSys. Other QueSys users are restored with equal access to the subsystem.
QueFreeze() prevents all other processes working within the QueSys, from proceeding with QueSys operations until a bracketing QueUnfreeze(), XipcUnfreeze() or XipcLogout() call is issued. The subsystems should therefore be kept frozen for as short a period of time as possible.
QueUnfreeze() will fail if the user has not previously frozen the QueSys via QueFreeze().

ERRORS

Code	Description
`QUE_ER_NOSUBSYSTEM`	QueSys is not configured in the instance.
`QUE_ER_NOTLOGGEDIN`	User not logged into instance (User never logged in, was aborted or disconnected).
`QUE_ER_NOTFROZEN`	QueSys not frozen.
`XIPCNET_ER_CONNECTLOST`	Connection to instance lost.
`XIPCNET_ER_NETERR`	Network transmission error.
`XIPCNET_ER_SYSERR`	Operating system error.

INTERACTIVE COMMAND

SYNTAX

queunfreeze

ARGUMENTS

None.

EXAMPLES

```
xipc> queunfreeze
      RetCode = 0
```

2.2.32 QueUnget() - UNGET A MESSAGE BACK TO A QUEUE

NAME

`QueUnget()` - Unget a Message Back To a Queue

SYNTAX

```
#include "xipc.h"


XINT
QueUnget(MsgHdr)

MSGHDR *MsgHdr;
```

PARAMETERS

Name	Description
`MsgHdr`	A pointer to a message header to be un-gotten.

RETURNS

Value	Description
RC >= 0	Unget successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueUnget() is used for returning a gotten message header to the queue it was taken from. QueUnget() returns the message header to its original position relative to other messages on the queue.

QueUnget() succeeds even if it must violate a queue's capacity by returning the message to the queue from which it was last taken. Refer to the QueSys/MemSys/SemSys User Guide for an explanation.

ERRORS

Code	Description
QUE_ER_BADQID	Bad Qid in *MsgHdr*.
QUE_ER_MSGHDRNOTREMOVED	*MsgHdr* references a message header that has not been dequeued.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
QUE_ER_CAPACITY_ASYNC_USER	QueSys async user table full.
QUE_ER_CAPACITY_HEADER	QueSys header table full.
QUE_ER_CAPACITY_NODE	QueSys node table full.

XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

queunget *MsgHdr*

ARGUMENTS

MsgHdr A one letter message header variable

EXAMPLES

```
xipc> queget f ea 0 wait
      RetCode = 0, Qid = 0, Seq# = 1211, Prio = 100, HdrStatus = REMOVED
xipc> quecopy f 0 *
      Text = "Mary had a little lamb"
xipc> queunget f
      RetCode = 0
```

2.2.33 QueUntrigger() - UNDEFINE A QUESYS TRIGGER

NAME

`QueUntrigger()` - Undefine a QueSys Trigger

SYNTAX

```
#include "xipc.h"


XINT
QueUntrigger(Sid, TriggerSpec)

XINT Sid;
... TriggerSpec;
```

PARAMETERS

Name	Description
`Sid`	The Semaphore Id of the event semaphore associated with the trigger to be undefined.
`TriggerSpec`	Specification of the QueSys trigger event to be deleted. The event is specified using a macro that defines the type of event and parameters such as *Qid* and threshold values. See the description part of QueTrigger() for a list of all triggers.

RETURNS

Value	Description
RC >= 0	QueUntrigger successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueUntrigger() is used to undefine a trigger that was previously defined using the QueTrigger() function.
The parameters to QueUntrigger() must be the same as were used to originally define the trigger.

ERRORS

Code	Description
`QUE_ER_BADQID`	*Qid* is not a valid queue ID.
`QUE_ER_BADSID`	*Sid* is not a valid semaphore ID.
`QUE_ER_BADTRIGGERCODE`	Bad trigger code.
`QUE_ER_BADUID`	*Uid* is not a valid user id.

QUE_ER_BADVAL	Illegal trigger parameter value.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
QUE_ER_TRIGGERNOTEXIST	Trigger not previously defined
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

queuntrigger *Sid TriggerCode TriggerParms*

ARGUMENTS

Sid	Semaphore Id used when the trigger was defined.
TriggerCode	Mnemonic code of the trigger. For a complete list of trigger codes, see the QueTrigger function in the QueSys Functions and Macros Chapter. Note that the prefix "QUE_T_" of the trigger code should not be specified, e.g., QUE_T_BYTES_HIGH should be specified as **bytes_high**.
TriggerParms	Additional parameters depending on the type of trigger defined.

EXAMPLES

```
xipc> quetrigger 31 bytes_high 7 80
      RetCode = 0
      .
      .
      .
xipc> queuntrigger 31 bytes_high 7 80
      RetCode = 0
```

2.2.34 QueWrite() - WRITE MESSAGE TEXT TO MESSAGE TEXT POOL

NAME

`QueWrite()` - Write Message Text To Message Text Pool

SYNTAX

```
#include "xipc.h"


XINT
QueWrite(MsgHdr, MsgBuf, MsgLength, Options)

MSGHDR *MsgHdr;
XANY *MsgBuf;
XINT MsgLength;
... Options;
```

PARAMETERS

Name	Description
`MsgHdr`	A pointer to an empty message header. *MsgHdr* values are assigned by QueWrite().
`MsgBuf`	A pointer to the message text to be written.
`MsgLength`	The size (in bytes) of the message in *MsgBuf*. Its value must be greater than 0.
`Options`	*Options* must be a valid *BlockOpt* option. See Appendix A, Using Blocking *X•IPC* Functions, for a description of *BlockOpt*.

RETURNS

Value	Description
RC >= 0	Write successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

QueWrite() attempts to write the text in *MsgBuf* to the message text pool. It then sets *MsgHdr* with appropriate values and returns it to the calling program. *MsgHdr* can then be placed on a queue using QuePut().
QueWrite() sets the message text pool reference count to one (1) when it creates a new text block.
QueWrite() is given the potential to block or complete asynchronously by setting *BlockOpt* appropriately. The operation will block or complete asynchronously if the message text pool currently lacks the capacity for a message text of size *MsgLength*.
A QueWrite() operation completes when the required free space becomes available in the text pool, usually by another user's actions. Specifically:

- Another user calling QueRead() or QueReceive() to remove a message's text from the message text pool.

See Appendix A, Using Blocking X•IPC Functions, for a description of how to use the blocking options.

ERRORS

Code	Description
QUE_ER_ASYNC	Operation is being performed asynchronously.
QUE_ER_ASYNCABORT	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
QUE_ER_BADBLOCKOPT	Invalid *BlockOpt*.
QUE_ER_BADBUFFER	*MsgBuf* is NULL.
QUE_ER_BADLENGTH	Invalid *MsgLength* parameter.
QUE_ER_BADOPTION	Invalid *Options* parameter.
QUE_ER_CAPACITY_ASYNC_USER	QueSys async user table full.
QUE_ER_CAPACITY_NODE	QueSys node table full.
QUE_ER_INTERRUPT	Operation was interrupted.
QUE_ER_ISFROZEN	A *BlockOpt* of QUE_WAIT or QUE_TIMEOUT() was specified after the instance was frozen by the calling user.
QUE_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.
QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
QUE_ER_NOWAIT	*BlockOpt* of QUE_NOWAIT specified and request was not immediately satisfied.
QUE_ER_TIMEOUT	The blocked QueWrite() operation timed out.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.
XIPCNET_ER_TOOBIG	Text exceeds instance's size limit.

INTERACTIVE COMMAND

SYNTAX

quewrite *MsgHdr MessageText BlockingOpt*

ARGUMENTS

MsgHdr A one letter message header variable.

MessageText The text of the message enclosed in double quotes.

BlockingOpt See the Blocking Options discussion in the `xipc` command (Interactive Command Processor) section at the beginning of this Manual.

EXAMPLES

```
xipc> quewrite a "Mary had a little lamb" wait
      RetCode = 0
xipc> queput a shq 0,1 100 wait
      RetCode = 0  Qid = 1
```

2.2.35 ADDITIONAL QUESYS INTERACTIVE COMMAND

msghdr - Display Message Header

SYNTAX

msghdr *MsgHdr*

ARGUMENTS

MsgHdr A one letter message header variable

EXAMPLES

```
xipc> msghdr a
      Qid = 1, Seq# = 1234, Prio = 1000, HdrStatus = REMOVED
```

2.3 Macros

2.3.1 MSGSELECTCODES - MESSAGE SELECT CODES USED FOR MESSAGE RETRIEVAL

NAME

`MsgSelectCodes` - Message Select Codes Used for Message Retrieval Operations

SYNTAX

```
#include "xipc.h"
```

DESCRIPTION

A QIDLIST, used as part of a message retrieval operation (QueGet() or QueReceive()), usually includes one or more Message Select Code macros *(MsgSelectCode*s) for designating each listed queue's candidate message.

The *MsgSelectCode*s that are acceptable as QIDLIST elements are:

`QUE_M_EA(`*`q`*`)`	Designates the earliest arrived (oldest) message on the queue *q*.
`QUE_M_LA(`*`q`*`)`	Designates the latest arrived (most recent) message on the queue *q*.
`QUE_M_HP(`*`q`*`)`	Designates the highest priority message on the queue *q*.
`QUE_M_LP(`*`q`*`)`	Designates the lowest priority message on the queue *q*.
`QUE_M_PREQ(`*`q, n`*`)`	Designates the first message on queue *q* having a priority of *n*.
`QUE_M_PRNE(`*`q, n`*`)`	Designates the first message on queue *q not* having a priority of *n*.
`QUE_M_PRGT(`*`q, n`*`)`	Designates the first message on queue *q* with a priority greater than *n*.
`QUE_M_PRGE(`*`q, n`*`)`	Designates the first message on queue *q* with a priority greater than or equal to *n*.
`QUE_M_PRLT(`*`q, n`*`)`	Designates the first message on queue *q* having a priority less than *n*.
`QUE_M_PRLE(`*`q, n`*`)`	Designates the first message on queue *q* with a priority less than or equal to *n*.
`QUE_M_PRRNG(`*`q,n,m`*`)`	Designates the first message on queue *q* with a priority in the range [*n*,*m*].
`QUE_M_SEQEQ(`*`q, seqn`*`)`	Designates the first message on queue *q* with a value equal to sequence number *seqn*.
`QUE_M_SEQGE (`*`q, seqn`*`)`	Designates the first message on queue *q* with a value greater than or equal to sequence number *seqn*.
`QUE_M_SEQLE(`*`q, seqn`*`)`	Designates the first message on queue *q* with a value less than or equal to sequence number *seqn*.
`QUE_M_SEQGT(`*`qid, seqn`*`)`	Designates the first message on queue *q* with a value greater than sequence number *seqn*.
`QUE_M_SEQLT(`*`q, seqn`*`)`	Designates the first message on queue *q* with a value less than sequence number *seqn*.

Note that *MsgSelectCode*s involving priorities cause the queue to be searched in decreasing priority order.

2.3.2 QUESELECTCODES - QUEUE SELECT CODES USED FOR MESSAGE DISPATCH AND RETRIEVAL

NAME

QueSelectCodes - Queue Select Codes Used for Message Dispatch and Retrieval Operations

SYNTAX

```
#include "xipc.h"
```

DESCRIPTION

A *QueSelectCode* is a required parameter for QuePut(), QueSend(), QueGet() and QueReceive(). The available *QueSelectCode*s depend on the function for which it is specified.

The *QueSelectCode*s that are valid within message dispatch functions (QuePut() and QueSend()) are:

QUE_Q_SHQ	Select the shortest queue.
QUE_Q_LNQ	Select the longest queue.
QUE_Q_HPQ	Select the queue having the highest priority message.
QUE_Q_LPQ	Select the queue having the lowest priority message.
QUE_Q_EAQ	Select the queue having the earliest arrived (oldest) message.
QUE_Q_LAQ	Select the queue with the latest arrived (most recent) message.
QUE_Q_ANY	Select the first queue in the list that has room (not full).

The *QueSelectCode*s that are valid within message retrieval functions (QueGet() and QueReceive()) can be divided into two groups:

Selection based on Message Attributes:

QUE_Q_EA	Select the earliest arrived (oldest) candidate message.
QUE_Q_LA	Select the latest arrived (most recent) candidate message.
QUE_Q_HP	Select the highest priority candidate message.
QUE_Q_LP	Select the lowest priority candidate message.

Selection based on Queue Attributes:

QUE_Q_LNQ	Select the candidate message from the longest queue in the list.
QUE_Q_SHQ	Select the candidate message from the shortest queue in the list.
QUE_Q_HPQ	Select the candidate message from the queue having the highest priority message.
QUE_Q_LPQ	Select the candidate message from the queue having the lowest priority message.
QUE_Q_EAQ	Select the candidate message from the queue having the earliest arrived message.
QUE_Q_LAQ	Select the candidate message from the queue having the latest arrived message.
QUE_Q_ANY	Select the first candidate message.

3. MEMSYS PARAMETERS, FUNCTIONS AND MACROS

3.1 *X•IPC* Instance Configuration - MemSys Parameters

NAME

***X•IPC* Instance Configuration** - MemSys parameter definitions for .cfg files

SYNTAX

```
[MEMSYS]
General MemSys parameters, defined below
```

PARAMETERS

The table below lists the general MemSys configuration parameters. Each parameter is presented with its name, description and default value. The order that parameters appear within the `[MEMSYS]` section of the configuration is not significant. The default values shown do *not* represent limits for the values that any particular user may require.

Parameter Name	**Description**	**Default Value**
`MAX_SEGMENTS`	The maximum number of concurrent segments. It should be set based on the requirements of the programs using the instance.	16
`MAX_USERS`	The maximum number of concurrent MemSys users (real users and pending asynchronous operations) that can be supported by the subsystem. It should be set based on the requirements of the programs using the instance. Note that asynchronously blocked MemSys operations are treated as MemSys users. The expected level of MemSys asynchronous activity should therefore be factored into this parameter.	32
`MAX_NODES`	The maximum number of nodes. MemSys nodes are used internally for tracking users that block on MemSys operations. The value depends largely on the nature of the program that will use the instance. A conservative estimate can be calculated with the following formula: **`MAX_NODES`**`=(`**`MAX_SEGMENTS`**`*`**`MAX_USERS`**`*` *`AverageSegmentSections`*`)+(`**`MAX_USERS`**`*4)+` **`MAX_SEGMENTS`**`)` where: *`AverageSegmentSections`* is the expected average number of sections that will exist concurrently on a segment. The default value was calculated using the default values for `MAX_SEGMENTS` and `MAX_USERS` and using the number *2*	1168

Parameter Name	Description	Default Value
	for ***AverageSegmentSections***.	
MAX_SECTIONS	The maximum expected number of sections that will exist concurrently in the instance. A starting value can be calculated with the following formula: **MAX_SECTIONS**=(**MAX_SEGMENTS***_AverageSegmentSections_) where: ***AverageSegmentSections*** is as defined above. The default value was calculated using the values indicated above.	32
SIZE_MEMPOOL	The size of the memory pool (K-bytes). **SIZE_MEMPOOL** must exceed the size of the largest segment that will be created in the instance. It must also exceed the largest aggregate of concurrent segments. A starting formula for **SIZE_MEMPOOL** is: **SIZE_MEMPOOL** = (**MAX_SEGMENTS***_AverageSegmentSize_) where: ***AverageSegmentSize*** is the expected average segment size occurring within the instance. The default value was calculated using the values indicated above and the number ***256*** for ***AverageSegmentSize.*** **SIZE_MEMPOOL** is expressed in terms of K-bytes. As such the calculated value should be rounded up o the next K-bytes multiple. (For example, if the calculation comes to 1948 bytes, then 2 K-bytes should be specified.)	4
SIZE_MEMTICK	The memory allocation unit (bytes). This value specifies the multiple by which memory pool allocations are made. **SIZE_MEMTICK** should be rounded up to a multiple of 4. A good starting value for **SIZE_MEMTICK** is: **SIZE_MEMTICK** = ***25PercentileSegmentSize*** where: ***25PercentileSegmentSize*** is the size value for which it is expected that 75% of the instance's segments will be larger in size and 25% will be smaller.	32

3.2 Functions

3.2.1 MemAbortAsync() - ABORT AN ASYNCHRONOUS OPERATION

NAME

`MemAbortAsync()` - Abort An Asynchronous Operation

SYNTAX

```
#include "xipc.h"


XINT
MemAbortAsync(AUid)

XINT AUid;
```

PARAMETERS

Name	Description
AUid	The asynchronous operation User ID of the operation to be aborted.

RETURNS

Value	Description
RC >= 0	Abort successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemAbortAsync() aborts a pending asynchronous operation.
If the aborted asynchronous operation was issued by the same *X•IPC* user, the *BlockOpt* of the aborted operation is ignored and the Asynchronous Result Control Block is not set.
If the aborted operation was issued by a different user, a return code of `MEM_ER_ASYNCABORT` is placed in the *RetCode* field of the operation's Asynchronous Result Control Block and the action specified in the *BlockOpt* of the aborted operation is carried out, i.e., a callback routine is invoked or a semaphore is set.

ERRORS

Code	Description
`MEM_ER_BADUID`	Invalid *AUid* parameter.
`MEM_ER_NOSUBSYSTEM`	MemSys is not configured in the instance.
`MEM_ER_NOTLOGGEDIN`	User not logged into instance (User never logged in, was aborted or disconnected).
`MEM_ER_SYSERR`	An internal error has occurred while processing the request.

XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memabortasync *AsyncUserId*

ARGUMENTS

AsyncUserId Asynchronous user id of the asynchronous MemSys operation to be aborted

EXAMPLES

```
xipc> memlock all (0 100 32) callback(cb1,m)
      RetCode = -1097
      Operation continuing asynchronously
xipc> acb m
      AUid = 35
      .
      .
xipc> memabortasync 35
......Callback function CB1 executing......
      RetCode = -1098
      Asynchronous operation aborted
      .
      .
      .
```

3.2.2 MemAccess() - ACCESS AN EXISTING MEMORY SEGMENT

NAME

`MemAccess()` - Access an Existing Memory Segment

SYNTAX

```
#include "xipc.h"


XINT
MemAccess(Name)

CHAR *Name;
```

PARAMETERS

Name	Description
Name	A pointer to a string that contains the symbolic name identifying the desired memory segment. *Name* must be null terminated, must not exceed `MEM_LEN_XIPCNAME` characters, must identify an existing memory segment and cannot be `MEM_PRIVATE`.

RETURNS

Value	Description
RC >= 0	Access successful. RC is memory segment ID (Mid). Mid is to be used in all subsequent MemSys calls that refer to this memory segment.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemAccess() accesses an existing memory segment in MemSys. *Name* is used for identifying the desired memory segment. The function returns the Mid of the accessed memory segment.

ERRORS

Code	Description
`MEM_ER_BADSEGNAME`	Invalid *Name* parameter.
`MEM_ER_NOSUBSYSTEM`	MemSys is not configured in the instance.
`MEM_ER_NOTFOUND`	Memory Segment with *Name* does not exist.
`MEM_ER_NOTLOGGEDIN`	User not logged into instance (User never logged in, was aborted or disconnected).
`XIPCNET_ER_CONNECTLOST`	Connection to instance lost.
`XIPCNET_ER_NETERR`	Network transmission error.
`XIPCNET_ER_SYSERR`	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memaccess *Name*

ARGUMENTS

Name Segment name

EXAMPLES

```
xipc> memaccess TrackTable
      Mid = 1
```

3.2.3 MemCreate() - CREATE A NEW MEMORY SEGMENT

NAME

`MemCreate()` - Create a New Memory Segment

SYNTAX

```
#include "xipc.h"


XINT
MemCreate(Name, Size)

CHAR *Name;
XINT Size;
```

PARAMETERS

Name	Description
Name	A pointer to a string that contains a symbolic name for publicly identifying the memory segment. *Name* must be null terminated and must not exceed MEM_LEN_XIPCNAME characters. If *Name* is MEM_PRIVATE then a private memory segment is created. Duplicate memory segment names (other than MEM_PRIVATE) are not permitted.
Size	The size of the memory segment to be created. The size is specified in units of bytes. *Size* must be greater than zero.

RETURNS

Value	Description
RC >= 0	Create successful. RC is memory segment ID (Mid). Mid is to be used in all subsequent MemSys calls that refer to this memory segment.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemCreate() creates a new memory segment in MemSys. *Name* is used for publicly identifying the new memory segment. A *Name* of MEM_PRIVATE directs MemSys to create a private memory segment (i.e., having no public identification). The segment is created having a size of *Size* bytes.

ERRORS

Code	Description
MEM_ER_BADSEGNAME	Invalid *Name* parameter.
MEM_ER_BADSIZE	Invalid *Size* parameter.
MEM_ER_CAPACITY_NODE	MemSys node table full.
MEM_ER_CAPACITY_POOL	MemSys text pool full.

MEM_ER_CAPACITY_SECTION	MemSys section table full.
MEM_ER_CAPACITY_TABLE	MemSys segment table full.
MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memcreate {*Name* | @PRIVATE} *Size*

ARGUMENTS

Name Segment name (or, if @PRIVATE, a private memory segment indicator).

Size Segment size

EXAMPLES

```
xipc> memcreate TrackTable 10240
      Mid = 1
```

3.2.4 MemDelete() - DELETE A MEMORY SEGMENT

NAME

`MemDelete()` - Delete a Memory Segment

SYNTAX

```
#include "xipc.h"


XINT
MemDelete(Mid)

XINT Mid;
```

PARAMETERS

Name	Description
Mid	The memory segment ID of the MemSys segment to be deleted. *Mid* was obtained by the user via MemCreate() or MemAccess() function calls.

RETURNS

Value	Description
RC >= 0	Delete successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemDelete() deletes the MemSys segment identified by *Mid* from MemSys. MemDelete() will fail if any sections are defined over the segment or if any user is blocked trying to lock, own, write or read the segment.

ERRORS

Code	Description
MEM_ER_BADMID	Invalid Memory Segment ID *Mid*.
MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
MEM_ER_MEMBUSY	MemSys Segment has one or more sections defined over it.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memdelete *Mid*

ARGUMENTS

Mid Segment Id

EXAMPLES

```
xipc> memdelete 1
      RetCode = 0
```

3.2.5 MemDestroy() - DESTROY A MEMSYS MEMORY SEGMENT

NAME

`MemDestroy()` - Destroy a MemSys Memory Segment

SYNTAX

```
#include "xipc.h"


XINT
MemDestroy(Mid)

XINT Mid;
```

PARAMETERS

Name	Description
Mid	The memory segment ID of the MemSys segment to be destroyed. *Mid* was obtained by the user via MemCreate() or MemAccess() function calls.

RETURNS

Value	Description
RC >= 0	Destroy successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemDestroy() deletes the MemSys segment identified by *Mid* from MemSys. MemDestroy() deletes the segment even if sections are currently defined over it or users are waiting to lock, own, write or read the segment.
Blocked MemSys operations (i.e., MemRead(), MemWrite(), MemLock() or MemSecOwn()), initiated by other users involving memory segment *Mid*, are interrupted and returned with an `RC = MEM_ER_DESTROYED`, indicating the forced deletion of memory segment *Mid*.
Users currently locking or owning section(s) defined over the destroyed memory segment have those sections silently removed from their possession. These users are not explicitly notified of the segment's forced deletion.

ERRORS

Code	Description
`MEM_ER_BADMID`	Invalid Memory Segment ID *Mid*.
`MEM_ER_NOSUBSYSTEM`	MemSys is not configured in the instance.
`MEM_ER_NOTLOGGEDIN`	User not logged into instance (User never logged in, was aborted or disconnected).

XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.
XIPCNET_ER_TOOBIG	Text exceed instance's size limit.

INTERACTIVE COMMAND

SYNTAX

memdestroy *Mid*

ARGUMENTS

Mid Segment Id

EXAMPLES

```
xipc> memdestroy 1
      RetCode = 0
```

3.2.6 MemFreeze() - FREEZE MEMSYS

NAME

`MemFreeze()` - Freeze MemSys

SYNTAX

```
#include "xipc.h"


XINT
MemFreeze()
```

PARAMETERS

None.

RETURNS

Value	Description
RC >= 0	MemFreeze successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemFreeze() freezes all MemSys activity occurring within the logged in instance and gives the calling user exclusive access to all MemSys functionality. MemSys remains frozen until a MemUnfreeze(), XipcUnfreeze() or an XipcLogout() function call is issued.
MemFreeze() prevents all other users, working within the MemSys, from proceeding with MemSys operations–until a bracketing MemUnfreeze(), XipcUnfreeze() or XipcLogout() is issued. The subsystem should therefore be kept frozen for as short a period of time as possible.

ERRORS

Code	Description
`MEM_ER_NOSUBSYSTEM`	MemSys is not configured in the instance.
`MEM_ER_NOTLOGGEDIN`	User not logged into instance (User never logged in, was aborted or disconnected).
`MEM_ER_ISFROZEN`	Calling user has already frozen MemSys.
`XIPCNET_ER_CONNECTLOST`	Connection to instance lost.
`XIPCNET_ER_NETERR`	Network transmission error.
`XIPCNET_ER_SYSERR`	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memfreeze

ARGUMENTS

None.

EXAMPLES

```
xipc> memfreeze
      RetCode = 0
```

3.2.7 MemInfoMem() - GET MEMORY SEGMENT INFORMATION

NAME

`MemInfoMem()` - Get Memory Segment Information

SYNTAX

```
#include "xipc.h"


XINT
MemInfoMem(Mid, InfoMem)

XINT Mid;
MEMINFOMEM *InfoMem;
```

PARAMETERS

Name	Description
Mid	The memory segment ID of the segment whose information is desired, or MEM_INFO_FIRST, or MEM_INFO_NEXT(*Mid*). *Mid* can be obtained via MemCreate() or MemAccess() function calls.
InfoMem	Pointer to a structure of type MEMINFOMEM, into which the segment information will be copied.

RETURNS

Value	Description
RC >= 0	Successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemInfoMem() fills the specified structure with information about the segment identified by *Mid*. The Mid argument can be specified as one of the following:

- *Mid* - a memory segment id identifying a specific memory segment
- MEM_INFO_FIRST - identifies the first valid memory segment id
- MEM_INFO_NEXT(*Mid*) - identifies the next valid memory segment id, following *Mid*.

A program reviewing the status of all queues within an instance should call MemInfoMem() specifying MEM_INFO_FIRST, followed by repeated calls to the function specifying MEM_INFO_NEXT until the MEM_ER_NOMORE error code is returned.

Each MemSys segment has two lists of information associated with it:

- SList: The list of sections currently defined over the specified memory segment. Each list element contains location, size, ownership and access privilege data about a section existing on the subject memory segment, at the time of the MemInfoMem call.

- WList: The list of blocked MemSys operations involving the specified memory segment. The operations are listed in the order that they blocked.

The MEMINFOMEM data structure follows:

```
/*
 * The MEMINFOMEM structure is used for retrieving status information
 * about a particular MemSys semaphore. MemInfoMem() fills the
 * structure with the data about the Mid it is passed.
 */

typedef struct _MEMINFOMEM
{
    XINT Mid;
    XINT CreateTime;                /* Time segment was created */
    XINT CreateUid;                 /* The Uid who created it */
    XINT Size;                          /* Size of segment (bytes)*/
    XINT NumSections;                   /* Num of sections on seg */
    XINT NumSecOwned;                   /* Num of owned sections  */
    XINT NumSecLocked;                  /* Num of locked sections */
    XINT NumBytesOwned;             /* Bytes owned on segment */
    XINT NumBytesLocked;            /* Bytes locked on segment */
    XINT CountWrite;                /* Num writes to segment  */
    XINT CountRead;                 /* Num reads from segment */
    XINT LastUidWrite;                  /* Last Uid to write segment */
    XINT LastUidRead;                   /* Last Uid to read segment */
    XINT LastUidOwned;                  /* Last Uid to own on segment */
    XINT LastUidLocked;             /* Last Uid to lock on segment */
    XINT SListTotalLength;
    XINT SListOffset;
    XINT SListLength;
    XINT WListTotalLength;
    XINT WListOffset;
    XINT WListLength;
    MEM_MEMSLISTITEM SList[MEM_LEN_INFOLIST];
    MEM_MEMWLISTITEM WList[MEM_LEN_INFOLIST];
    CHAR Name[MEM_LEN_XIPCNAME + 1]; /* Segment name */
}
MEMINFOMEM;
```

where:

SListTotalLength returns with the total internal length of the SList for this segment.

SListOffset is set by the user, prior to the MemInfoMem() function call, to specify the portion of the SList that should be returned (i.e. what offset to start from).

SListLength returns with the length of the SList portion returned by the current call to MemInfoMem(). More specifically, *SListLength* is the number of elements returned in the *SList* array. *SListLength* will be between 0 and `MEM_LEN_INFOLIST`.

SList is an array of list elements, where each element is of type `MEM_MEMSLISTITEM`. The `MEM_MEMSLISTITEM` data type is defined in mempubd.h. The data structure follows:

```
typedef struct _MEM_MEMSLISTITEM
{
    XINT OwnerUid;
    XINT OwnerPriv;
    XINT OtherPriv;
    XINT Mid;
    XINT Offset;
    XINT Size;
}
MEM_MEMSLISTITEM;

typedef struct _MEM_MEMWLISTITEM
{
    XINT Uid;
    XINT OpCode;                /* MEM_BLOCKEDLOCK, MEM_BLOCKEDREAD,
                                 * MEM_BLOCKEDWRITE or MEM_BLOCKEDOWN
                                 */
    XINT Offset;
    XINT Size;
}
MEM_MEMWLISTITEM;
```

Similar definitions and usage rules apply to the WList related fields.
A call to MemInfoMem() should be preceded by the setting of the *SListOffset* and *WListOffset* fields of the MEMINFOMEM structure to appropriate values.
For a full example and more complete information on using the Info functions, refer to the Advanced Topics chapter of the *X♦IPC* User Guide.

ERRORS

Code	**Description**
MEM_ER_BADMID	No segment with specified *Mid*.
MEM_ER_BADLISTOFFSET	Invalid offset value specified.
MEM_ER_NOMORE	No more memory segments.
MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

meminfomem *Mid* | **first** | **next(***Mid***)**

ARGUMENTS

Mid Print info on the **first** segment, the segment with Mid *MeId* or the **next** higher segment.

EXAMPLES

```
xipc> meminfomem first
      Name: 'TrackTable'
      Sections Defined: 0       Bytes Allocated: 10240
      . . .

xipc> meminfomem next(1)
      Mid: 3 Name: 'NodeTable'
      Sections Defined: 4       Bytes Allocated: 10240
      . . .

xipc> meminfomem next(3)
      RetCode = -1038
      No more data in list id
```

3.2.8 MemInfoSec() - GET SECTION INFORMATION

NAME

`MemInfoSec()` - Get Section Information

SYNTAX

```
#include "xipc.h"


XINT
MemInfoSec(Section, InfoSec)

SECTION Section;
MEMINFOSEC *InfoSec;
```

PARAMETERS

Name	Description
`Section`	The section whose status is requested.
`InfoSec`	Pointer to a structure of type `MEMINFOSEC`, into which the section information will be copied.

RETURNS

Value	Description
RC >= 0	Successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemInfoSec() fills the specified structure with information about the section identified by *Section*. The data structure follows:

```
/*
 * The MEMINFOSEC structure is used for retrieving status information
 * about a particular MemSys section overlay. MemInfoSec() fills the
 * structure with the data about the Section it is passed.
 */

typedef struct _MEMINFOSEC
{
    XINT Mid;               /* MemSys segment ID */
    XINT Offset;        /* Offset into the segment */
    XINT Size;              /* Section size in bytes */
    XINT OwnerUid;          /* Uid of section owner */
    XINT OwnerPriv;         /* Owner access privileges */
    XINT OtherPriv;         /* Other access privileges */
}
MEMINFOSEC;
```

For a full example and more complete information on using the Info functions, refer to the Advanced Topics chapter of the X♦IPC User Guide.

ERRORS

Code	Description
MEM_ER_BADSECTION	Invalid *Section* parameter.
MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

meminfosec *Section*

ARGUMENTS

Section Either a one letter section variable or section descriptor: Mid, Offset and Size enclosed in parentheses.

EXAMPLES

```
xipc> memsecdef (1 100 64)
      RetCode = 0

xipc> memsection a (1 100 64)
      Section = (1 100 64)

xipc> meminfosec a
      Mid: 1, Offset: 100, Size: 64
      Owner: 32
      . . .

xipc> meminfosec (1 100 64)
      Mid: 1, Offset: 100, Size: 64
      Owner: 32
      . . .
```

3.2.9 MemInfoSys() - GET SUBSYSTEM INFORMATION

NAME

`MemInfoSys()` - Get System Information

SYNTAX

```
#include "xipc.h"


XINT
MemInfoSys(InfoSys)

MEMINFOSYS *InfoSys;
```

PARAMETERS

Name	Description
`InfoSys`	Pointer to a structure of type `MEMINFOSYS`, into which the system information will be copied.

RETURNS

Value	Description
RC >= 0	Successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemInfoSys() fills the specified structure with information about the current instance of MemSys into which the user is logged in. The data structure follows:

```
/*
 * The MEMINFOSYS structure is used for retrieving status information
 * about the MemSys instance. MemInfoSys() fills the structure with the
 * data about the instance.
 */

typedef struct _MEMINFOSYS
{
    XINT MaxUsers;                      /* Max configured users */
    XINT CurUsers;                      /* Current num of users */
    XINT MaxSegments;                   /* Max configured segments */
    XINT CurSegments;                   /* Current num of segments */
    XINT MaxNodes;                      /* Max configured nodes */
    XINT FreeNCnt;                      /* Current available nodes */
    XINT MaxSections;                   /* Max configured sections */
    XINT FreeSCnt;                      /* Current available sects */
    XINT MemPoolSizeBytes;              /* Configured mem pool size */
    XINT MemTickSize;                   /* Configured mem tick size */
    XINT MemPoolTotalAvail;          /* Free text pool space */
```

```
    XINT MemPoolLargestBlk;          /* Largest contig block */
    XINT MemPoolMaxPosBlks;          /* Max possible tick blocks */
    XINT MemPoolTotalBlks;               /* Number allocated blocks */
    CHAR Name[MEM_LEN_PATHNAME + 1]; /* InstanceFilename */
}
MEMINFOSYS;
```

For a full example and more complete information on using the Info functions, refer to the Advanced Topics chapter of the X•IPC User Guide.

ERRORS

Code	Description
MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

meminfosys

ARGUMENTS

None.

EXAMPLES

```
xipc> meminfosys
      Configuration:  '/usr/config'
      .......  Maximum  Current
      Users:        60       11
      .
      .
      .
```

3.2.10 MemInfoUser() - GET USER MEMSYS INFORMATION

NAME

`MemInfoUser()` - Get User Information

SYNTAX

```
#include "xipc.h"


XINT
MemInfoUser(Uid, InfoUser)

XINT Uid;
MEMINFOUSER *InfoUser;
```

PARAMETERS

Name	Description
`Uid`	The user ID of the user whose information is desired, or `MEM_INFO_FIRST,` or `MEM_INFO_NEXT(`*`Uid`*`).` *Uid* may be an asynchronous Uid (AUid).
`InfoUser`	Pointer to a structure of type `MEMINFOUSER`, into which the user information will be copied.

RETURNS

Value	Description
RC >= 0	Successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemInfoUser() fills the specified structure with information about the user identified by *Uid*. The Uid argument can be specified as one of the following:

- *`Uid`* - an integer user ID identifying a specific user
- `MEM_INFO_FIRST` - identifies the first valid user ID within the instance
- `MEM_INFO_NEXT(`*`Uid`*`)` - identifies the next valid user ID, following *`Uid`*.

A program reviewing the status of all users currently within MemSys would call MemInfoUser() specifying `MEM_INFO_FIRST,` followed by repeated calls to the function specifying `MEM_INFO_NEXT` until the `MEM_ER_NOMORE` error code is returned.

Each MemSys user has three lists of information associated with it:

- ☐ HList: The list of sections currently held (owned or locked) by the subject user. The sections are listed in the order that they were acquired.
- ☐ QList: The list of sections currently being requested by the subject user. The QList will have elements only when the user is blocked on a MemSecOwn() or MemLock() operation.

- WList: The list of sections currently being waited on by the subject user. The WList is the subset of the QList that has not yet been satisfied. It too will only have elements when the user is blocked on a MemSecOwn() or MemLock() operation.

The MEMINFOUSER data structure follows:

```
/*
 * The MEMINFOUSER structure is used for retrieving status information
 * about a particular MemSys user. MemInfoUser() fills the structure
 * with the data about the Uid it is passed.
 */

typedef struct _MEMINFOUSER
{
    XINT Uid;
    XINT Pid;                          /* Process Id of user  */
    TID  Tid;                          /* Thread ID of user  */
    XINT LoginTime;                 /* Time of login to MemSys */
    XINT TimeOut;                      /* Remaining timeout secs */
    XINT WaitType;                     /* One of: MEM_BLOCKEDWRITE,
                                        * MEM_BLOCKEDREAD, MEM_BLOCKEDOWN,
                                        * MEM_BLOCKEDLOCK or MEM_USER_NOTWAITING
                                        */
    XINT NumSecOwned;                  /* Num sects owned by Uid */
    XINT NumSecLocked;                 /* Num sects locked by Uid */
    XINT NumBytesOwned;             /* Num bytes owned by Uid */
    XINT NumBytesLocked;            /* Num bytes locked by Uid */
    XINT CountWrite;                /* Num of Uid write opers */
    XINT CountRead;                 /* Num of Uid read opers  */
    XINT HListTotalLength;
    XINT HListOffset;
    XINT HListLength;
    XINT WListTotalLength;
    XINT WListOffset;
    XINT WListLength;
    XINT QListTotalLength;
    XINT QListOffset;
    XINT QListLength;
    MEM_USERHLISTITEM HList[MEM_LEN_INFOLIST];
    MEM_USERWLISTITEM WList[MEM_LEN_INFOLIST];
    MEM_USERQLISTITEM QList[MEM_LEN_INFOLIST];
    CHAR Name[MEM_LEN_XIPCNAME + 1]; /* User login name /
    CHAR NetLoc[XIPC_LEN_NETLOC + 1];   /* Name of Client Node */
}
MEMINFOUSER;
```

where:

HListTotalLength returns with the total internal length of the HList for this user.

HListOffset is set by the user, prior to the MemInfoUser() function call, to specify the portion of the HList that should be returned (i.e. what offset to start from).

HListLength returns with the length of the HList portion returned by the current call to MemInfoUser(). More specifically, *HListLength* is the number of elements returned in the *HList* array. *HListLength* will be between 0 and MEM_LEN_INFOLIST.

HList is an array of list elements, where each element is of type MEM_USERHLISTITEM. The MEM_USERHLISTITEM data type is defined in mempubd.h. The data structure follows.

Similar definitions and usage rules apply to the QList and WList related fields.

```
typedef struct _MEM_USERHLISTITEM
{
    XINT OpCode;                    /* MEM_BLOCKEDLOCK or MEM_BLOCKEDOWN */
    XINT Mid;
    XINT Offset;
    XINT Size;
}
MEM_USERHLISTITEM;

typedef struct _MEM_USERWLISTITEM
{
    XINT Mid;
    XINT Offset;
    XINT Size;
}
MEM_USERWLISTITEM;

typedef struct _MEM_USERQLISTITEM
{
    XINT Mid;
    XINT Offset;
    XINT Size;
}
MEM_USERQLISTITEM;
```

A call to MemInfoUser() should be preceded by the setting of the *HListOffset*, *QListOffset* and *WListOffset* fields of the MEMINFOUSER structure to appropriate values.
For a full example and more complete information on using the Info functions, refer to the Advanced Topics chapter of the *X♦IPC* User Guide.

ERRORS

Code	**Description**
MEM_ER_BADUID	Invalid *Uid* parameter.
MEM_ER_BADLISTOFFSET	Invalid offset value specified.
MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOMORE	No more users.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

meminfouser *UserId* | **first** | **next(***UserId***)** | **all**

ARGUMENTS

UserId Print info on the **first** user, the user with Uid *Uid* or the **next** higher user.

QueId

EXAMPLES

```
xipc> meminfouser 9
      Name: 'TableManager'  Pid: 241  Tid: 0
      Login Time:...
      .
      .
      .
```

3.2.11 MemList(), MemListBuild() - BUILD LISTS OF MEMORY SECTIONS

NAME

`MemList()` - Create a One-Time List of Memory Sections

`MemListBuild()` - Build a Reusable List of Memory Sections

SYNTAX

```
#include "xipc.h"


PMIDLIST
MemList(Sec1, Sec2, ..., MEM_EOL)

SECTION Sec1;
SECTION Sec2;
...

PMIDLIST
MemListBuild(MidList, Sec1, Sec2, ..., MEM_EOL)

MIDLIST MidList;
SECTION Sec1;
SECTION Sec2;
...
```

PARAMETERS

Name	Description
`MidList`	An area to contain the resultant MIDLIST_xe "MIDLIST"_. A pointer to a MIDLIST (type PMIDLIST) may be passed as well.
`Sec1, Sec2`	Memory sections to be included in the resultant MIDLIST. `MEM_EOL_xe "MEM_EOL"_` must be used to mark the end of the list.

RETURNS

Value	Description
RC != `NULL`	A pointer to the created list of sections. For MemListBuild() it is a pointer to the *MidList* specified as an argument. For MemList() it is a pointer to an internal *MidList*.
RC == `NULL`	*MidList* exceeded `MEM_LEN_MIDLIST` elements.

DESCRIPTION

These functions are used for building lists of memory section in a format acceptable by MemSys functions taking a MIDLIST as one of their arguments. `MEM_EOL` must be the last argument to MemList() and MemListBuild().
MemListBuild() builds the list in the area specified by *MidList*. MemList() creates the list in an internal static area, and can therefore be safely used only once.

ERRORS

None.

3.2.12 MemListAdd(), MemListRemove() - UPDATE LIST OF MEMORY SECTIONS

NAME

`MemListAdd()` - Create a One-Time List of Sids

`MemListRemove()` - Build a Reusable List of Sids

SYNTAX

```
#include "xipc.h"

PMIDLIST
MemListAdd(MidList, Sec1, Sec2, ..., MEM_EOL)

MIDLIST MidList;
SECTION Sec1;
SECTION Sec2;
...

PMIDLIST
MemListRemove(MidList, Sec1, Sec2, ..., MEM_EOL)

MIDLIST MidList;
SECTION Sec1;
SECTION Sec2;
...
```

PARAMETERS

Name	Description
MidList	The MIDLIST to be updated_xe "QIDLIST"_. A pointer to a MIDLIST (type PMIDLIST) may be passed as well.
Sec1, Sec2,	The memory sections to be added to or removed from the MIDLIST. `MEM_EOL_xe "QUE_EOL"_` must be used to mark the end of the argument list.

RETURNS

Value	Description
RC != `NULL`	A pointer to the updated MIDLIST specified as an argument.
RC == `NULL`	The operation failed. The MIDLIST specified as an argument remains unchanged.

DESCRIPTION

These functions are used for modifying MIDLISTs by adding or removing memory sections.
MEM_EOL must be the last argument to MemListAdd() and MemListRemove().
MemListAdd() adds memory sections to an existing MIDLIST. The new memory sections are added at the end of the specified MidList. If the number of memory sections being added, plus the current number of memory sections in the MIDLIST, exceeds MEM_LEN_MIDLIST, then the operation fails, NULL is returned and Mdlist remains unchanged.
MemListRemove() removes memory sections from an existing MIDLIST. Each memory section must match an memory section of the MidList exactly, and then that memory section is removed. If it does not match a memory section of the MidList, the operation fails.
If the operation succeeds, a pointer to the modified argument MidList is returned; otherwise NULL is returned and the argument MidList remains unchanged.

ERRORS

None.

3.2.13 MemListCount() – GET NUMBER OF SECTIONS IN A LIST OF SECTIONS

NAME

MemListCount() - Get Number of Sections in a List of Sections

SYNTAX

```
#include "xipc.h"

XINT
MemListCount(MidList)

MIDLIST MidList;
```

PARAMETERS

Name	Description
MidList	A MIDLIST or a pointer to a MIDLIST (type PMIDLIST).

RETURNS

Value	Description
RC < 0	The MIDLIST is invalid.
RC > = 0	Number of sections in MidList.

DESCRIPTION

MemListCount() is used for determining the number of sections contained in MIDLIST.

ERRORS

None.

3.2.14 MemLock() - LOCK MEMORY SECTION(S)

NAME

`MemLock()` - Lock Memory Section(s)

SYNTAX

```
#include "xipc.h"


XINT
MemLock(LockType, MidList, RetSec, Options)

XINT LockType;
MIDLIST MidList;
SECTION *RetSec;
... Options;
```

PARAMETERS

Name	Description
`LockType`	`MEM_ANY`, `MEM_ALL` or `MEM_ATOMIC` depending on the locking criteria desired.
`MidList`	A list of memory section(s) to be locked. This list can be a MIDLIST built by MemList() or MemListBuild(). A pointer to a MIDLIST (type PMIDLIST) may be passed as well.
`RetSec`	A pointer to a section variable that gets assigned a value by MemLock() upon its return. It is acceptable to have a null *RetSec* argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired. Successful lock operations (`RC >= 0`) return with **RetSec* identifying the last locked section. Interrupted lock operations, where `RC = MEM_ER_DESTROYED`, return with **RetSec* identifying the destroyed section. Failed calls with `RC = MEM_ER_BADSECTION` return with **RetSec* identifying the invalid section. **RetSec* is otherwise undefined.
`Options`	*Options* must be a valid *BlockOpt* option. See Appendix A, Using Blocking *X•IPC* Functions, for a description of *BlockOpt*.

RETURNS

Value	Description
RC >= 0	MemLock successful. If *`LockType`* `= MEM_ANY` then one of the requested memory sections has been locked and **RetSec* identifies the locked memory section. If *`LockType`* `= MEM_ALL` or `MEM_ATOMIC` then all requested memory sections have been locked by the calling user and **RetSec* identifies the last memory section that was locked.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemLock() attempts to lock the memory sections in *MidList* for the calling user's exclusive read-write access, based on the values of *LockType* and *BlockOpt*. The value of *LockType* specifies how to satisfy the requested lock:

- If `LockType = MEM_ANY`, the request is considered satisfied when any one of the memory sections in *MidList* has been locked.
- If `LockType = MEM_ALL`, the request is not considered satisfied until all the memory sections in *MidList* have been locked. Memory sections are locked as they become available until all the sections in *MidList* have been locked.
- If `LockType = MEM_ATOMIC`, the request is not considered satisfied until all the memory sections in *MidList* are available and then locked in a single atomic operation. Individual memory sections in *MidList* are not locked as they become available. In this way it differs from `LockType = MEM_ALL`.

MemLock() will not succeed in locking a section unless all of the segment bytes overlaid by the section are read and write accessible by the calling user.
The value of *BlockOpt* specifies whether the requested lock operation should block or complete asynchronously. See Appendix A, Using Blocking X•IPC Functions, for a description of how to use the blocking options.
A blocked call that is interrupted by an asynchronous signal or trap is returned with `RC = MEM_ER_INTERRUPT`. Satisfied MemLock() calls return the identity of the last locked memory section in **RetSec*. It is acceptable to have a null *RetSec* argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired.
Calling MemLock() is the most direct approach for acquiring exclusive read-write control over one or more memory sections. MemLock() first defines the *MidList* list of memory sections that do not yet exist. It then acquires ownership rights over the sections as specified by *LockType*. It then sets their read-write privilege to read-write by the caller and non-accessible by others. In this way MemLock() is a concatenation of MemSecDef(), MemSecOwn() and MemSecPriv().

ERRORS

Code	Description
`MEM_ER_ALREADYLOCKED`	*MidList* contains a memory section that is already locked by the user. **SecPtr* identifies the invalid sectionl
`MEM_ER_ASYNC`	Operation is being performed asynchronously.
`MEM_ER_ASYNCABORT`	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
`MEM_ER_BADBLOCKOPT`	Invalid *BlockOpt*.
`MEM_ER_BADOPTION`	Invalid *Options* parameter.
`MEM_ER_BADLOCKTYPE`	Invalid *LockType* parameter.
`MEM_ER_BADSECTION`	*MidList* contains a bad section. **RetSec* is set to the invalid section.
`MEM_ER_CAPACITY_ASYNC_USER`	MemSys user table full.
`MEM_ER_CAPACITY_NODE`	MemSys node table full.
`MEM_ER_CAPACITY_SECTION`	MemSys section table full.
`MEM_ER_DESTROYED`	Another user destroyed a memory section that was being waited on by this user. The blocked lock operation was cancelled. No memory sections were locked. **RetSec* identifies the destroyed memory section.

MEM_ER_ISFROZEN	A *BlockOpt* of MEM_WAIT or MEM_TIMEOUT() was specified after the instance was frozen by the calling user.
MEM_ER_INTERRUPT	The blocked lock operation was interrupted by an asynchronous event (such as a signal). The operation has been canceled.
MEM_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.
MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
MEM_ER_NOWAIT	*BlockOpt* of MEM_NOWAIT was specified and the request was not immediately satisfied.
MEM_ER_TIMEOUT	The time out period for the blocked lock operation has expired without satisfying the request.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memlock *LockType MidList BlockingOpt*

ARGUMENTS

LockType **any**, **all** or **atomic**.

MidList List of section variables or memory section descriptors separated by commas. Each descriptor consists of Mid, Offset and Size enclosed in parentheses.

BlockingOpt See the Blocking Options discussion in the xipc command (Interactive Command Processor) section at the beginning of this Manual.

EXAMPLES

```
xipc> memsection s (1 0 20)
      Section = (1 0 20)
xipc> memlock atomic (0 120 30),(0 4000 30),s wait
      RetCode = 0
      Section = (1 0 20)
```

3.2.15 MemPointer() - GET POINTER TO MEMSYS SEGMENT

NAME

MemPointer() - Get Pointer to MemSys Segment

SYNTAX

```
#include "xipc.h"


XINT
MemPointer(Mid, Ptr);

XINT Mid;
XANY **Ptr;
```

PARAMETERS

Name	Description
Mid	The memory segment ID of the MemSys segment whose pointer is desired. *Mid* was obtained by the user via MemCreate() or MemAccess() function calls.
Ptr	A pointer to the pointer variable that is returned with the segment pointer.

RETURNS

Value	Description
RC >= 0	MemPointer successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemPointer() returns with a pointer to the first byte (offset 0) of MemSys segment *Mid*. The pointer can then be used for directly accessing the data within the segment. The pointer is returned via the *Ptr* argument.

This function is a double-edged sword. On the one hand, it provides the most basic mode of access to a MemSys segment. This can simplify certain coding tasks. On the other hand, using a direct pointer into a MemSys segment for manipulating its data completely circumvents the software synchronization and access control mechanisms inherent to MemWrite() and MemRead(). It also introduces the risk of overrunning MemSys segment boundaries.

As such, a direct segment pointer should only be used (if at all) on areas of a MemSys segment that are currently "locked" by the User. To use otherwise will produce unpredictable results at best.

MemPointer() will return with a valid pointer to a segment of MemSys instance, if the instance involved is local to the calling program (not over the network). Requests for pointers to MemSys segments regarding instances that are non-local return with a `MEM_ER_NOTLOCAL` error code.

ERRORS

Code	Description
MEM_ER_BADBUFFER	*Ptr* is NULL.
MEM_ER_BADMID	Invalid Memory Segment ID *Mid*.
MEM_ER_NOTLOCAL	Instance is not local.
MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).

INTERACTIVE COMMAND

SYNTAX

mempointer *Mid*

ARGUMENTS

Mid Segment Id

EXAMPLES

```
xipc> memcreate TrackTable 10240
      Mid = 1
xipc> mempointer 1
      Pointer = 0001A000
```

3.2.16 MemRead() - READ DATA FROM A MEMORY SEGMENT

NAME

MemRead() - Read Data From a Memory Segment

SYNTAX

```
#include "xipc.h"


XINT
MemRead(Mid, Offset, Length, Buffer, Options)

XINT Mid;
XINT Offset;
XINT Length;
XANY *Buffer;
... Options;
```

PARAMETERS

Name	Description
Mid	The memory segment ID of the segment to be read. *Mid* was obtained by the user via MemCreate() or MemAccess() function calls.
Offset	The number of bytes beyond the start of the segment from where the MemRead operation should commence.
Length	The number of bytes to be read from the memory segment into *Buffer*. Its value must be greater than 0.
Buffer	A pointer to the target data buffer.
Options	*Options* must be a valid *BlockOpt* option. See Appendix A, Using Blocking X•IPC Functions, for a description of *BlockOpt*.

RETURNS

Value	Description
RC >= 0	MemRead successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemRead() attempts to read *Length* bytes of data from the MemSys Segment identified by *Mid*, starting at *Offset* bytes from the start of the segment, into *Buffer*. The MemRead() operation will succeed if and only if all the bytes targeted by the operation are currently readable by the calling user. MemRead() operations are guaranteed to be atomic.

If the entire target area is protected from other user access (i.e. all overlaying sections are locked by the reading user),then the read operation is executed in its most efficient form, without the need for explicit protection by MemSys.
If, however, any part of the target area is not protected from other user access (i.e. one or more of the overlaying sections are not locked by the calling user), then the atomic nature of the read operation is explicitly enforced by MemSys.
MemRead() is given the potential to block or complete asynchronously by setting *BlockOpt* appropriately. The operation will block or complete asynchronously if any bytes of the target area are not readable by the calling user. A MemRead() operation completes when the complete target memory area becomes readable by the caller. This is usually accomplished by another user's calling MemUnlock() or MemSecPriv() regarding the unreadable section(s) overlaying the target area.
See Appendix A, Using Blocking *X•IPC* Functions, for a description of how to use the blocking options.

ERRORS

Code	**Description**
MEM_ER_ASYNC	Operation is being performed asynchronously.
MEM_ER_ASYNCABORT	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
MEM_ER_BADBLOCKOPT	Invalid *BlockOpt*.
MEM_ER_BADBUFFER	*Buffer* is NULL.
MEM_ER_BADMID	Invalid Memory Segment ID *Mid*.
MEM_ER_BADOPTION	Invalid *Options* parameter.
MEM_ER_BADTARGET	Invalid target specification.
MEM_ER_CAPACITY_ASYNC_USER	MemSys async user table full.
MEM_ER_CAPACITY_NODE	MemSys node table full.
MEM_ER_DESTROYED	Another user destroyed the memory segment targeted by the blocked MemRead operation.
MEM_ER_INTERRUPT	Operation was interrupted.
MEM_ER_ISFROZEN	A *BlockOpt* of MEM_WAIT or MEM_TIMEOUT() was specified after the instance was frozen by the calling user.
MEM_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.
MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
MEM_ER_NOWAIT	*BlockOpt* of MEM_NOWAIT specified and request was not immediately satisfied.
MEM_ER_TIMEOUT	The blocked MemRead() operation timed out.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memread *Mid Offset Length BlockingOpt*

ARGUMENTS

Mid Segment Id.

Offset Number of bytes beyond the start of the segment.

Length Number of bytes

BlockingOpt See the Blocking Options discussion in the `xipc` command (Interactive Command Processor) section at the beginning of this Manual.

EXAMPLES

```
xipc> memread 1 0 22 wait
      Text = "Mary had a little lamb"
```

3.2.17 MemSecDef() - DEFINE A MEMORY SECTION

NAME

`MemSecDef()` - Define a Memory Section

SYNTAX

```
#include "xipc.h"

XINT
MemSecDef(Section)

SECTION Section;
```

PARAMETERS

Name	Description
`Section`	The section to be defined. *Section* identifies the location and size of the new memory section.

RETURNS

Value	Description
RC >= 0	MemSecDef successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemSecDef() defines a new memory section on a MemSys memory segment. Attempting to define a section that already exists is an error.

Read and Write access to each MemSys segment is governed via memory sections that overlay the segment. Each MemSys segment can have zero or more sections defined over it. A section influences access to the underlying portion of the segment using a pair of access privilege mechanisms: *owner* read-write privilege and *other* read-write privilege.

A user may become the owner of a section, at which time his section access privilege is governed via the section's *owner* access privilege setting. Otherwise, users are treated as *others* (non-owners) with their section access privilege governed via the *other* privilege setting. A section can be owned by at most one user at a time.

A section that is owned by a user and that has its *other* privileges set to `MEM_NA` is considered *Locked* by the user. (Refer to the MemLock() function call.)

MemRead() and MemWrite() operations do not succeed unless the calling user has the appropriate access to every byte within the targeted area of the MemSys segment. Such a determination depends on the sections defined over the targeted segment area, the read-write privilege of these sections and relevant section ownerships (if any) at the time.

A newly-defined section has no owner, and has `MEM_RW` (read and write) privilege set for *owner* and *other*s.

ERRORS

Code	Description
MEM_ER_BADSECTION	Invalid *Section* parameter.
MEM_ER_CAPACITY_SECTION	MemSys section table full.
MEM_ER_DUPLICATE	Section already exists.
MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memsecdef *Section*

ARGUMENTS

Section Either a one letter section variable or section descriptor: Mid, Offset and Size enclosed in parentheses.

EXAMPLES

```
xipc> memsecdef (1 100 64)
      RetCode = 0
```

3.2.18 MemSecOwn() - BECOME OWNER OF MEMORY SECTION(S)

NAME

`MemSecOwn()` - Become Owner of Memory Section(s)

SYNTAX

```
#include "xipc.h"


XINT
MemSecOwn(OwnType, MidList, RetSec, Options)

XINT OwnType;
MIDLIST MidList;
SECTION *RetSec;
... Options;
```

PARAMETERS

Name	Description
`OwnType`	`MEM_ANY, MEM_ALL or MEM_ATOMIC` depending on the criteria desired.
`MidList`	A list of section(s) to be owned. This list can be a MIDLIST built by MemList() or MemListBuild(). A pointer to a MIDLIST (type PMIDLIST) may be passed as well.
`RetSec`	A pointer to a `SECTION` variable that gets assigned a value by MemSecOwn() upon its return. It is acceptable to have a null *RetSec* argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired. Successful operations (RC >= 0) return with **RetSec* identifying the last acquired section. Interrupted operations, where `RC = MEM_ER_DESTROYED`, return with **RetSec* identifying the destroyed section. Failed calls with `RC = MEM_ER_BADSECTION` return with **RetSec* identifying the invalid section. The value of **RetSec* is otherwise undefined.
`Options`	*Options* must be a valid *BlockOpt* option. See Appendix A, Using Blocking *X•IPC* Functions, for a description of *BlockOpt*.

RETURNS

Value	Description
RC >= 0	MemSecOwn successful. If *`OwnType`* `= MEM_ANY` then one of the requested memory sections has been acquired and **RetSec* identifies the memory section. If *OwnType* = `MEM_ALL` or `MEM_ATOMIC` then all requested memory sections have been acquired by the calling user and **RetSec* identifies the last memory section gotten.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemSecOwn() attempts to become the owner of the memory section(s) listed in *MidList*, based on the value of *OwnType* and *BlockOpt*.
MemSecOwn() assumes that the listed sections are already defined. It does not define any new sections. Section definition can be accomplished using MemSecDef().
When MemSecOwn() becomes owner of the listed sections, it does not modify the sections' *owner* or *other* read-write privileges settings in any manner. This can be accomplished using MemSecPriv().
MemSecOwn() will not succeed in becoming the owner of a section unless all of the bytes overlaid by the section are read and write accessible by the calling user.

The value of *OwnType* specifies how the ownership request is satisfied:

- If `OwnType = MEM_ANY`, the request is considered satisfied when any one of the memory sections in *MidList* has been acquired.
- If `OwnType = MEM_ALL`, the request is not considered satisfied until all the memory sections in *MidList* have been acquired. Memory sections are acquired as they become available until all the sections in *MidList* are owned.
- If `OwnType = MEM_ATOMIC`, the request is not considered satisfied until all the memory sections in *MidList* are available and then acquired in a single atomic operation. Individual memory sections in *MidList* are not acquired as they become available. In this way it differs from `OwnType = MEM_ALL`.

The value of *BlockOpt* specifies whether the operation should block or complete asynchronously. See Appendix A, Using Blocking X•IPC Functions, for a description of how to use the blocking options.
A blocked call that is interrupted by an asynchronous signal or trap is returned with `RC = MEM_ER_INTERRUPT`. Satisfied MemSecOwn() calls return the identity of the last acquired memory section in **RetSec*. It is acceptable to have a null *RetSec* argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired.
The most direct approach for acquiring exclusive read-write access to a list of memory sections is to use the MemLock()function call. Refer to the MemLock() function for further details on section locking.

ERRORS

Code	Description
`MEM_ER_ASYNC`	Operation is being performed asynchronously.
`MEM_ER_ASYNCABORT`	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
`MEM_ER_BADBLOCKOPT`	Invalid *BlockOpt*.
`MEM_ER_BADOPTION`	Invalid *Options* parameter.
`MEM_ER_BADOWNTYPE`	Invalid *OwnType* parameter.
`MEM_ER_BADMIDLIST`	Invalid MidList parameter.
`MEM_ER_BADSECTION`	*MidList* contains a bad section. **RetSec* is set to the invalid section.
`MEM_ER_CAPACITY_ASYNC_USER`	MemSys async user table full.
`MEM_ER_CAPACITY_NODE`	MemSys node table full.
`MEM_ER_DESTROYED`	Another user destroyed a memory section that was being waited on by this user. The blocked MemSecOwn operation was cancelled. No memory sections were acquired. **RetSec* identifies the destroyed memory section.
`MEM_ER_INTERRUPT`	The blocked MemSecOwn operation was interrupted by an asynchronous event (such as a signal). The operation has

	been cancelled.
MEM_ER_ISFROZEN	A *BlockOpt* of MEM_WAIT or MEM_TIMEOUT() was specified after the instance was frozen by the calling user.
MEM_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.
MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
MEM_ER_NOWAIT	BlockOpt of MEM_NOWAIT was specified and the request was not immediately satisfied.
MEM_ER_TIMEOUT	The time out period for the blocked MemSecOwn operation has expired without satisfying the request.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memsecown *OwnType MidList BlockingOpt*

ARGUMENTS

LockType **any**, **all** or **atomic**.

MidList List of section variables or memory section descriptors separated by commas. Each descriptor consists of Mid, Offset and Size enclosed in parentheses.

BlockingOpt See the Blocking Options discussion in the xipc command (Interactive Command Processor) section at the beginning of this Manual.

EXAMPLES

```
xipc> memsecown all (0 120 30),(0 4000 30) post(35,b)
      RetCode = -1097
      Operation continuing asynchronously
xipc> semwait all 35 timeout(30)
      RetCode = 0  Sid = 35
xipc> acb b
      AUid            = 11
      .
      .
      .
      MEMSECOWN
      - RetSec  = (0 4000 30)
      - RetCode = 0
```

3.2.19 MemSecPriv() - SET A MEMORY SECTION'S PRIVILEGES

NAME

MemSecPriv() - Set a Memory Section's Privileges

SYNTAX

```
#include "xipc.h"


XINT
MemSecPriv(Section, OwnerPrivilege, OtherPrivilege)

SECTION Section;
XINT OwnerPrivilege;
XINT OtherPrivilege;
```

PARAMETERS

Name	Description
Section	The section whose privileges are to be set. *Section* identifies the location and size of the memory section involved.
OwnerPrivilege	The Section's *owner* privilege setting code.
OtherPrivilege	The Section's *other* privilege setting code.

RETURNS

Value	Description
RC >= 0	MemSecPriv successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemSecPriv() sets the read-write privileges settings of the given memory section. It sets the section's *owner* and *other* access privilege according to the values of *OwnerPrivilege* and *OtherPrivilege*. Possible privilege values are:

MEM_RW	Set privilege to read-write access.
MEM_RO	Set privilege to read only access.
MEM_WO	Set privilege to write only access.
MEM_NA	Set privilege to no access.
MEM_ADD_R	Add read access.
MEM_ADD_W	Add write access.
MEM_RMV_R	Remove read access.
MEM_RMV_W	Remove write access.
MEM_NC	Leave access privilege unchanged.

MemSecPriv() will fail if the calling user is not the current owner of the specified memory section. MemSecOwn() or MemLock() must first be used to acquire ownership of the section.

ERRORS

Code	Description
MEM_ER_BADSECTION	Invalid *Section* parameter.
MEM_ER_BADPRIVILEGE	Invalid *OwnerPrivilege* or *OtherPrivilege* parameter(s).
MEM_ER_NOTOWNER	User is not current owner of *Section*.
MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memsecpriv *Section OwnerPriv OtherPriv*

ARGUMENTS

Section Either a one letter section variable or section descriptor: Mid, Offset and Size enclosed in parentheses.

OwnerPriv Owner Privileges: **rw, ro, wo, na, +r, +w, -r, -w, nc.**

OtherPriv Others Privileges: **rw, ro, wo, na, +r, +w, -r, -w, nc.**

EXAMPLES

```
xipc> memsection s (1 100 64)
      Section = (1 100 64)
xipc> memsecdef s
      RetCode = 0
xipc> memsecown all s wait
      RetCode = 0
xipc> memsecpriv s rw ro
      RetCode = 0
      .
      .
      .
xipc> memsecpriv s -w nc
      RetCode = 0
```

3.2.20 MemSecRel() - RELEASE OWNED MEMORY SECTION(S)

NAME

MemSecRel() - Release Owned Memory Section(s)

SYNTAX

```
#include "xipc.h"


XINT
MemSecRel(MidList, RetSec)

MIDLIST MidList;
SECTION *RetSec;
```

PARAMETERS

Name	Description
MidList	A list of sections being released. This list can be a MIDLIST built by MemList() or MemListBuild(). A pointer to a MIDLIST (type PMIDLIST) may be passed as well.
RetSec	A pointer to a variable that gets assigned a value by MemSecRel() on return. It is acceptable to have a null *RetSec* argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired. For failed calls, where `RC = MEM_ER_BADSECTION` or `RC = MEM_ER_SECNOTOWNER`, **RetSec* identifies the invalid section. For successful calls **RetSec* identifies the last memory section released. *RetSec is otherwise undefined.

RETURNS

Value	Description
RC >= 0	MemSecRel successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemSecRel() releases the memory section(s) specified in the *MidList*. MemSecRel() will fail if any of the listed sections are not currently owned by the calling user.

It is acceptable to have a null *RetSec* argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired.

ERRORS

Code	Description
MEM_ER_BADSECTION	*MidList* contains a bad section. **RetSec* identifies the invalid section.
MEM_ER_BADMIDLIST	Invalid MidList parameter.

MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
MEM_ER_NOTOWNER	*MidList* contains a memory section not currently owned by the user. **RetSec* identifies the invalid section.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memsecrel *MidList*

ARGUMENTS

MidList List of section variables or memory section descriptors separated by commas. Each descriptor consists of Mid, Offset and Size enclosed in parentheses.

EXAMPLES

```
xipc> memsecown any (0 120 30),(0 4000 30) wait
      RetCode = 0
      Section = (0 120 30)
      .
      .
      .
xipc> memsecrel (0 120 30)
      RetCode = 0
```

3.2.21 MemSection() - INITIALIZE A SECTION VARIABLE

NAME

MemSection() - Initialize a Section Variable

SYNTAX

```
#include "xipc.h"


SECTION
MemSection(Mid, Offset, Size)

XINT Mid;
XINT Offset;
XINT Size;
```

PARAMETERS

Name	Description
Mid	The Memory Segment Id of the section. *Mid* was obtained by the user via MemCreate() or MemAccess() function calls.
Offset	The number of bytes beyond the start of the segment from where the section starts.
Size	The size of the memory section. The size is specified in units of bytes.

RETURNS

See description.

DESCRIPTION

MemSection() constructs a memory section variable having the specified initialized values, and returns the initialized variable as the function's value. As such, it can be used anywhere a SECTION variable is expected.

Note

MemSection() is not reentrant and should not be used in an environment that requires reentrant code, e.g., threaded environment or an environment that serves software interrupts such as signals, ASTs, etc. The function MemSectionBuild() should be used instead.

ERRORS

None.

INTERACTIVE COMMAND

SYNTAX

memsection *SectionId [Section]*

ARGUMENTS

SectionId A one letter section variable.

Section Either a one letter section variable or section descriptor: Mid, Offset and Size enclosed in parentheses.

EXAMPLES

```
xipc> memsection s (0 0 100)
      Section = (0 0 100)
xipc> section s
      Section = (0 0 100)
xipc> memsecdef s
      RetCode = 0
```

3.2.22 MemSectionBuild() - BUILD A SECTION VARIABLE

NAME

MemSectionBuild() - Build a Section Variable

SYNTAX

```
#include "xipc.h"


SECTION *
MemSectionBuild(SectionPtr, Mid, Offset, Size)

SECTION *SectionPtr;
XINT Mid;
XINT Offset;
XINT Size;
```

PARAMETERS

Name	Description
SectionPtr	A pointer to the section variable to be built.
Mid	The memory segment ID of the section. *Mid* was obtained by the user via MemCreate() or MemAccess() function calls.
Offset	The number of bytes beyond the start of the segment from where the section starts.
Size	The size of the memory section. The size is specified in units of bytes.

RETURNS

See description.

DESCRIPTION

MemSectionBuild() initializes the memory section variable specified in its first argument to have the specified values; it returns a pointer to the initialized variable as the function's value.

ERRORS

None.

3.2.23 MemSecUndef() - UNDEFINE A MEMORY SECTION

NAME

`MemSecUndef()` - Undefine a Memory Section

SYNTAX

```
#include "xipc.h"


XINT
MemSecUndef(Section)

SECTION Section;
```

PARAMETERS

Name	Description
`Section`	The identity of the section to be undefined.

RETURNS

Value	Description
RC >= 0	MemSecUndef successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemSecUndef() removes a memory section definition from a MemSys memory segment. The section argument identifies the section definition to be removed.
Undefining a section can have the affect of eliminating access barriers to all or part of the MemSys segment area it overlays.
A user can only undefine a section that he owns or that is unowned. Otherwise the call returns with `RC = MEM_ER_ACCESSDENIED`.
Users blocked trying to acquire ownership of the section being undefined are notified of the section's removal via a `RC = MEM_ER_DESTROYED` return code.

ERRORS

Code	Description
`MEM_ER_BADSECTION`	Invalid *Section* parameter.
`MEM_ER_NOSUBSYSTEM`	MemSys is not configured in the instance.
`MEM_ER_NOTLOGGEDIN`	User not logged into instance (User never logged in, was aborted or disconnected).
`MEM_ER_ACCESSDENIED`	Specified *Section* is currently owned by another user.

XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memsecundef *Section*

ARGUMENTS

Section Either a one letter section variable or section descriptor: Mid, Offset and Size enclosed in parentheses and separated by spaces.

EXAMPLES

```
xipc> memsecundef (1 100 64)
      RetCode = 0
```

3.2.24 MemTrigger() - DEFINE A MEMSYS TRIGGER

NAME

`MemTrigger()` - Define a MemSys Trigger

SYNTAX

```
#include "xipc.h"


XINT
MemTrigger(Sid, TriggerSpec)

XINT Sid;
... TriggerSpec;
```

PARAMETERS

Name	Description
Sid	The Semaphore ID of the event semaphore to be set when the trigger event occurs. The *Sid* is obtained by SemCreate() or SemAccess() function calls.
TriggerSpec	Specification of the MemSys trigger event. The event is specified using a macro that defines the type of event and parameters such as *Mid* and threshold values. See the description below for a list of all trigger specifications.

RETURNS

Value	Description
RC >= 0	MemTrigger successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

A MemSys trigger is a logical link between a MemSys event and a SemSys event semaphore. The semaphore is automatically set when the MemSys event occurs.
A trigger is defined by:

- ☐ The Id of an event semaphore that will be set when the MemSys event occurs.
- ☐ A MemSys event specification.

The *Sid* of the event semaphore is obtained by calling SemSys functions SemCreate() or SemAccess().
The following table contains a list of all MemSys events that can be specified:

Trigger	**Description**
`MEM_T_READ(Mid, Offset, Size)`	Trigger event when data is read from the shared memory area specified by *Mid*, *Offset* and *Size* (or any part of it).
`MEM_T_WRITE(Mid, Offset, Size)`	Trigger event when data is written into the shared memory area specified by *Mid*, *Offset* and *Size*

	(or any part of it).
MEM_T_LOCK(*Mid, Offset, Size*)	Trigger event when the shared memory area specified by *Mid*, *Offset* and *Size* (or any part of it) is locked.
MEM_T_UNLOCK(*Mid, Offset, Size*)	Trigger event when the shared memory area specified by *Mid*, *Offset* and *Size* (or any part of it) is unlocked.
MEM_T_USER_READ(*Mid, Offset, Size, Uid*)	Trigger event when user *Uid* reads data from the shared memory area specified by *Mid*, *Offset* and *Size* (or any part of it).
MEM_T_USER_WRITE(*Mid, Offset, Size, Uid*)	Trigger event when user *Uid* writes data into the shared memory area specified by *Mid*, *Offset* and *Size* (or any part of it).
MEM_T_USER_LOCK(*Mid, Offset, Size, Uid*)	Trigger event when user *Uid* locks the shared memory area specified by *Mid*, *Offset* and *Size* (or any part of it).
MEM_T_USER_UNLOCK(*Mid, Offset, Size, Uid*)	Trigger event when user *Uid* unlocks the shared memory area specified by *Mid*, *Offset* and *Size* (or any part of it).
MEM_T_POOL_HIGH(*N*)	Trigger event when the allocated size of the shared memory pool becomes higher than *N* percent of its capacity.
MEM_T_POOL_LOW(*N*)	Trigger event when the allocated size of the shared memory pool becomes lower than *N* percent of its capacity.
MEM_T_SECTION_HIGH(*N*)	Trigger event when the number of allocated sections becomes higher than *N* percent of the capacity.
MEM_T_SECTION_LOW(*N*)	Trigger event when the number of allocated sections becomes lower than *N* percent of the capacity.

ERRORS

Code	Description
MEM_ER_BADMID	*Mid* is not a valid segment ID.
MEM_ER_BADSID	*Sid* is not a valid semaphore ID.
MEM_ER_BADTRIGGERCODE	Bad trigger code
MEM_ER_BADUID	*Uid* is not a valid user ID.
MEM_ER_BADVAL	Illegal trigger parameter value
MEM_ER_BADTARGET	Illegal shared memory offset
MEM_ER_CAPACITY_NODE	MemSys node table full.
MEM_ER_DUPLICATE	Attempt to define a trigger that is already defined
MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).

XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memtrigger *Sid TriggerCode TriggerParms*

ARGUMENTS

Sid Semaphore Id of the semaphore to be set when the MemSys trigger event occurs.

TriggerCode Mnemonic code of the trigger. Note that the prefix 'MEM_T_" of the trigger code should not be specified, e.g., MEM_T_POOL_HIGH should be specified as **pool_high**.

TriggerParms Additional parameters depending on the type of trigger defined.

EXAMPLES

```
xipc> memcreate TrackTable 10240
      Mid = 1
xipc> semcreate BytesHighSem clear
      Sid = 31
xipc> # Set Semaphore 31 when any byte in the 10k Segment 1 is locked.
xipc> memtrigger 31 lock 1 0 10240
      RetCode = 0
```

3.2.25 MemUnfreeze() - UNFREEZE MEMSYS

NAME
`MemUnfreeze()` - Unfreeze MemSys

SYNTAX
```
#include "xipc.h"


XINT
MemUnfreeze()
```

PARAMETERS
None.

RETURNS

Value	Description
RC >= 0	MemUnfreeze successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION
MemUnfreeze() unfreezes MemSys. Other MemSys users are restored with equal access to the subsystem.
MemFreeze() prevents all other processes working within the MemSys, from proceeding with MemSys operations until a bracketing MemUnfreeze(), XipcUnfreeze() or XipcLogout() call is issued. The subsystems should therefore be kept frozen for as short a period of time as possible.
MemUnfreeze() will fail if the user has not previously frozen the MemSys via MemFreeze().

ERRORS

Code	Description
`MEM_ER_NOSUBSYSTEM`	MemSys is not configured in the instance.
`MEM_ER_NOTLOGGEDIN`	User not logged into instance (User never logged in, was aborted or disconnected).
`MEM_ER_NOTFROZEN`	MemSys not frozen.
`XIPCNET_ER_CONNECTLOST`	Connection to instance lost.
`XIPCNET_ER_NETERR`	Network transmission error.
`XIPCNET_ER_SYSERR`	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memunfreeze

ARGUMENTS

None.

EXAMPLES

```
xipc> memunfreeze
      RetCode = 0
```

3.2.26 MemUnlock() - UNLOCK MEMORY SECTION(S)

NAME

MemUnlock() - Unlock Memory Section(s)

SYNTAX

```
#include "xipc.h"


XINT
MemUnlock(MidList, RetSec)

MIDLIST MidList;
SECTION *RetSec;
```

PARAMETERS

Name	Description
MidList	A list of sections being unlocked. This list can be a MIDLIST built by MemList() or MemListBuild(). A pointer to a MIDLIST (type PMIDLIST) may be passed as well.
RetSec	A pointer to a variable that gets assigned a value by MemUnlock() on return. It is acceptable to have a null *RetSec* argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired. For failed calls, where `RC = MEM_ER_BADSECTION` or `RC = MEM_ER_NOTLOCKED`, **RetSec* identifies the invalid section. For successful calls **RetSec* identifies the last memory section unlocked. **RetSec* is otherwise undefined.

RETURNS

Value	Description
RC >= 0	MemUnlock successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemUnlock() unlocks the memory section(s) specified in the *MidList*. MemUnlock() will fail if any of the listed sections are not currently locked by the user (`RC = MEM_ER_NOTLOCKED`).

It is acceptable to have a null RetSid argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired.

ERRORS

Code	Description
MEM_ER_BADSECTION	*MidList* contains a bad section. **RetSec* identifies the invalid section.
MEM_ER_BADMIDLIST	Invalid MidList parameter.
MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
MEM_ER_NOTLOCKED	*MidList* contains a memory section not currently locked by the user. **RetSec* identifies the invalid section.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memunlock *MidList*

ARGUMENTS

MidList List of section variables or memory section descriptors separated by commas. Each descriptor consists of Mid, Offset and Size enclosed in parentheses.

EXAMPLES

```
xipc> memunlock (0 120 30),(0 4000 30)
      RetCode = 0
```

3.2.27 MemUntrigger() - UNDEFINE A MEMSYS TRIGGER

NAME

`MemUntrigger()` - Undefine a MemSys Trigger

SYNTAX

```
#include "xipc.h"


XINT
MemUntrigger(Sid, TriggerSpec)

XINT Sid;
... TriggerSpec;
```

PARAMETERS

Name	Description
`Sid`	The Semaphore Id of the event semaphore associated with the trigger to be undefined.
`TriggerSpec`	Specification of the MemSys trigger to be undefined. The event is specified using a macro that defines the type of event and parameters such as *Mid* and threshold values. See the description part of MemTrigger() for a list of all triggers.

RETURNS

Value	Description
RC >= 0	MemUntrigger successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemUntrigger() is used to undefine a trigger that was previously defined using the MemTrigger() function.
The parameters to MemUntrigger() must be the same as were used to originally define the trigger.

ERRORS

Code	Description
`MEM_ER_BADMID`	*Mid* is not a valid segment ID.
`MEM_ER_BADSID`	*Sid* is not a valid semaphore ID.
`MEM_ER_BADTRIGGERCODE`	Bad trigger code
`MEM_ER_BADUID`	*Uid* is not a valid user id.
`MEM_ER_BADVAL`	Illegal trigger parameter value
`MEM_ER_NOSUBSYSTEM`	MemSys is not configured in the instance.

MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
MEM_ER_TRIGGERNOTEXIST	Trigger not previously defined
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

memuntrigger *Sid TriggerCode TriggerParms*

ARGUMENTS

Sid	Semaphore Id used when the trigger was defined.
TriggerCode	Mnemonic code of the trigger. Note that the prefix 'MEM_T_" of the trigger code should not be specified, e.g., MEM_T_POOL_HIGH should be specified as **pool_high**.
TriggerParms	Additional parameters depending on the type of trigger defined.

EXAMPLES

```
xipc> memtrigger 31 lock 1 0 10240
      RetCode = 0
      .
      .
      .
xipc> memuntrigger 31 lock 1 0 10240
      RetCode = 0
```

3.2.28 MemWrite() - WRITE DATA INTO A MEMORY SEGMENT

NAME

`MemWrite()` - Write Data Into a Memory Segment

SYNTAX

```
#include "xipc.h"


XINT
MemWrite(Mid, Offset, Length, Buffer, Options)

XINT Mid;
XINT Offset;
XINT Length;
XANY *Buffer;
... Options;
```

PARAMETERS

Name	Description
`Mid`	The Memory Segment Id of the segment to be written. *Mid* was obtained by the user via MemCreate() or MemAccess() function calls.
`Offset`	The number of bytes beyond the start of the segment from where the MemWrite operation should commence.
`Length`	The number of bytes to be written from *Buffer* to the MemSys segment starting at *Offset* bytes into the segment. If `MEM_FILL()` is the *Buffer* argument then *Length* is the number of bytes to be filled. *Length* must be greater than 0.
`Buffer`	A pointer to a source data buffer or a call to the `MEM_FILL`() macro. `MEM_FILL`() when specified, is used for filling each byte of the targeted memory area with a specific byte value. `MEM_FILL`() takes the desired byte fill value as its argument.
`Options`	*Options* must be a valid *BlockOpt* option. See Appendix A, Using Blocking X•IPC Functions, for a description of *BlockOpt*.

RETURNS

Value	Description
RC >= 0	MemWrite successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

MemWrite() attempts to write *Length* bytes of data from *Buffer* into the memory segment identified by *Mid* starting at *Offset* bytes from the start of the segment. If `MEM_FILL`() is given as the *Buffer* argument, then the specified byte value is written to the entire targeted memory area.
The MemWrite operation will succeed if and only if all the bytes targeted by the operation are currently writeable by the calling user.
MemWrite operations are guaranteed to be atomic.
If the entire target area is protected from other user access (i.e., all overlaying sections are locked by the writing user), then the atomic write operation is executed in its most efficient form, without the need for explicit protection by MemSys.
If, however, any part of the targeted area is not protected from other user access (i.e., one or more of the overlaying sections are not locked by the calling user), then the atomic nature of the write operation is explicitly enforced by MemSys.
MemWrite() is given the potential to block or complete asynchronously by setting *BlockOpt* appropriately. The operation will block or complete asynchronously if any bytes of the target area cannot be written to by the calling user.
A MemWrite() operation completes when the complete target memory area becomes writeable by the caller. This is usually accomplished by another user's calling MemUnlock() or MemSecPriv() regarding the un-writeable section(s) underlying the target area.
See Appendix A, Using Blocking *X•IPC* Functions, for a description of how to use the blocking options.

ERRORS

Code	Description
`MEM_ER_ASYNC`	Operation is being performed asynchronously.
`MEM_ER_ASYNCABORT`	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
`MEM_ER_BADBLOCKOPT`	Invalid *BlockOpt*.
`MEM_ER_BADBUFFER`	*Buffer* is `NULL`.
`MEM_ER_BADMID`	Invalid Memory Segment ID *Mid*.
`MEM_ER_BADOPTION`	Invalid *Options* parameter.
`MEM_ER_BADTARGET`	Invalid target specification.
`MEM_ER_CAPACITY_ASYNC_USER`	MemSys async user table full.
`MEM_ER_CAPACITY_NODE`	MemSys node table full.
`MEM_ER_DESTROYED`	Another user destroyed the memory segment targeted by the blocked MemWrite operation.
`MEM_ER_INTERRUPT`	Operation was interrupted.
`MEM_ER_ISFROZEN`	A *BlockOpt* of `MEM_WAIT` or `MEM_TIMEOUT`() was specified after the instance was frozen by the calling user.
`MEM_ER_NOASYNC`	An asynchronous operation was attempted with no asynchronous environment present.
`MEM_ER_NOSUBSYSTEM`	MemSys is not configured in the instance.
`MEM_ER_NOTLOGGEDIN`	User not logged into instance (User never logged in, was aborted or disconnected).
`MEM_ER_NOWAIT`	*BlockOpt* of `MEM_NOWAIT` specified and request was not immediately satisfied.
`MEM_ER_TIMEOUT`	The blocked MemWrite() operation timed out.

XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.
XIPCNET_ER_TOOBIG	Text exceed instance's size limit.

INTERACTIVE COMMAND

SYNTAX

memwrite *Mid Offset Length Text BlockingOpt*

ARGUMENTS

Mid	Segment Id.
Offset	Number of bytes beyond the start of the segment.
Length	Number of bytes
Text	The text to be written enclosed in double quotes or a filler character enclosed in single quotes
BlockingOpt	See the Blocking Options discussion in the `xipc` command (Interactive Command Processor) section at the beginning of this Manual.

EXAMPLES

```
xipc> memwrite 1 0 22 "Mary had a little lamb" wait
      RetCode = 0

xipc> # Fill first 22 bytes of segment 1 with spaces.
xipc> memwrite 1 0 22 ' ' wait
      RetCode = 0
```

3.2.29 ADDITIONAL MEMSYS INTERACTIVE COMMAND

section - Display a Section Variable

SYNTAX

section *SectionId*

ARGUMENTS

SectionId A one letter section variable

EXAMPLES

```
xipc> section a
      Section = (0  100  25)
```

4. SEMSYS PARAMETERS, FUNCTIONS AND MACROS

4.1 X•IPC Instance Configuration - SemSys Parameters

NAME

X•IPC Instance Configuration - SemSys parameter definitions for .cfg files

SYNTAX

```
[SEMSYS]
General SemSys parameters, defined below
```

PARAMETERS

The table below lists the general SemSys configuration parameters, including the parameter name, description and default value. The order in which parameters appear within the `[SEMSYS]` section of the configuration is not significant. The default values shown do *not* represent limits for the values that any particular user may require.

Parameter Name	Description	Default Value
MAX_SEMS	The number of concurrent semaphores. It should be set based on the requirements of the programs using the instance.	16
MAX_USERS	The maximum number of concurrent SemSys users (real users and pending asynchronous operations) that can be supported by the subsystem. It should be set based on the requirements of the programs using the instance. Note that asynchronously blocked SemSys operations are treated as SemSys users. The expected level of SemSys asynchronous activity should therefore be factored into this parameter.	32
MAX_NODES	The maximum number of nodes. It defines the number of nodes that are to be made available to the instance. SemSys nodes are used internally for recording blocking and ownership of the instance's semaphores. There is no hard and fast rule for calculating an appropriate value for **MAX_NODES**. It depends on the mix of event vs. Resource semaphores to be employed, the number of user programs involved and the degree of blocking that is expected. An initial calculation can be made with the following formula: **MAX_NODES** = (**MAX_SEMS** * 2) + (**MAX_USERS** * 4) + (**MAX_USERS***MAX_SEMS**)** The default value was calculated using the default values for	640

Parameter Name	Description	Default Value
	MAX_SEMS and **MAX_USERS**. Empirical observations via SemView should be made to monitor node usage. Adjustments should follow as necessary.	

4.2 Functions

4.2.1 SemAbortAsync() - ABORT AN ASYNCHRONOUS OPERATION

NAME

SemAbortAsync() - Abort An Asynchronous Operation

SYNTAX

```
#include "xipc.h"


XINT
SemAbortAsync(AUid)

XINT AUid;
```

PARAMETERS

Name	Description
AUid	The asynchronous operation User ID of the operation to be aborted.

RETURNS

Value	Description
RC >= 0	Abort successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemAbortAsync() aborts a pending asynchronous operation.
If the aborted asynchronous operation was issued by the same X•IPC user, the *BlockOpt* of the aborted operation is ignored and the Asynchronous Result Control Block is not set.
If the aborted operation was issued by a different user, a return code of SEM_ER_ASYNCABORT is placed in the *RetCode* field of the operation's Asynchronous Result Control Block and the action specified in the *BlockOpt* of the aborted operation is carried out, i.e., a callback routine is invoked or a semaphore is set.

ERRORS

Code	Description
SEM_ER_BADUID	Invalid *AUid* parameter.
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
SEM_ER_SYSERR	An internal error has occurred while processing the request.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

semabortasync *AsyncUserId*

ARGUMENTS

AsyncUserId Asynchronous user id of the asynchronous SemSys operation to be aborted

EXAMPLES

```
xipc> semwait all 0 callback(cb1,s)
      RetCode = -1097
      Operation continuing asynchronously
      Sid = 0
xipc> acb s
      AUid = 35
      .
      .
xipc> semabortasync 35
......Callback function CB1 executing......
      RetCode = -1098
      Asynchronous operation aborted
      RetSid = 0
      RetCode = 0
```

4.2.2 SemAccess() - ACCESS AN EXISTING SEMAPHORE

NAME

SemAccess() - Access an Existing Semaphore

SYNTAX

```
#include "xipc.h"


XINT
SemAccess(Name)

CHAR *Name;
```

PARAMETERS

Name	Description
Name	A pointer to a string that contains the symbolic name identifying the desired semaphore. *Name* must be null terminated, must not exceed SEM_LEN_XIPCNAME characters, must identify an existing semaphore and cannot be SEM_PRIVATE.

RETURNS

Value	Description
RC >= 0	Access successful. RC is semaphore ID (Sid). Sid is to be used in all subsequent SemSys calls that refer to this semaphore.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemAccess() accesses an existing semaphore in SemSys. *Name* is used for identifying the desired semaphore. The function returns the *Sid* of the accessed semaphore.

ERRORS

Code	Description
SEM_ER_BADSEMNAME	Invalid *Name* parameter.
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTFOUND	Semaphore with *Name* does not exist.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

semaccess *Name*

ARGUMENTS

Name Semaphore name

EXAMPLES

```
xipc> semaccess InitSem
      Sid = 7
```

4.2.3 SemAcquire() - ACQUIRE RESOURCE SEMAPHORES

NAME

SemAcquire() - Acquire Resource Semaphores

SYNTAX

```
#include "xipc.h"


XINT
SemAcquire(AcquireType, SidList, RetSid, Options)

XINT AcquireType;
SIDLIST SidList;
XINT *RetSid;
... Options;
```

PARAMETERS

Name	Description
AcquireType	SEM_ANY, SEM_ALL or SEM_ATOMIC depending on the acquisition criteria desired.
SidList	A list of Sids being requested. This list should be a SIDLIST built by SemList() or SemListBuild() and updated by SemListAdd(). A pointer to a SIDLIST (type PSIDLIST) may be passed as well.
RetSid	A pointer to a variable that gets assigned by SemAcquire() upon its return. It is acceptable to have a null RetSid argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired. Successful acquire operations (RC >= 0) return with **RetSid* equal to the last acquired Sid. Interrupted acquire operations (RC = SEM_ER_DESTROYED or SEM_ER_CANCEL) return with **RetSid* equal to the destroyed or cancelled Sid. Failed calls with RC = SEM_ER_BADSID return with **RetSid* equal to the invalid Sid. **RetSid* is otherwise undefined.
Options	*Options* must be a valid *BlockOpt* option. See Appendix A, Using Blocking X•IPC Functions, for a description of *BlockOpt*.

RETURNS

Value	Description
RC >= 0	Acquire successful. If *AcquireType* = SEM_ANY then one of the requested semaphores has been acquired and **RetSid* is the Sid of the acquired semaphore. If *AcquireType* = SEM_ALL or SEM_ATOMIC then all requested semaphores have been acquired by the calling user and **RetSid* is the Sid of the last semaphore acquired.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemAcquire() attempts to obtain the semaphores in *SidList* for the calling user based on the values of *AcquireType* and *BlockOpt*.
The value of *AcquireType* specifies how to satisfy the requested acquire:

- If *AcquireType* = SEM_ANY, the request is considered satisfied when any one of the semaphores in *SidList* has been acquired.
- If *AcquireType* = SEM_ALL, the request is not considered satisfied until all the semaphores in *SidList* have been acquired. Semaphores are accumulated as they become available until the entire *SidList* has been acquired.
- If *AcquireType* = SEM_ATOMIC, the request is not considered satisfied until all the semaphores in *SidList* are available and then acquired in a single atomic operation. Individual semaphores in *SidList* are not accumulated as they become available. In this way it differs from *AcquireType* = SEM_ALL.

The value of *BlockOpt* specifies whether and how to block for the satisfaction of the requested acquire in case it cannot be satisfied immediately. See Appendix A, Using Blocking *X•IPC* Functions, for a description of how to use the blocking options.
A blocked call that is interrupted by an asynchronous signal or trap is returned with RC = SEM_ER_INTERRUPT.
Satisfied SemAcquire() calls return the Sid of the last acquired semaphore in **RetSid*. It is acceptable to have a null RetSid argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired.

ERRORS

Code	Description
SEM_ER_ASYNC	Operation is being performed asynchronously.
SEM_ER_ASYNCABORT	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
SEM_ER_BADACQUIRETYPE	Invalid *AcquireType* parameter.
SEM_ER_BADBLOCKOPT	Invalid *BlockOpt*.
SEM_ER_BADOPTION	Invalid *Options* parameter.
SEM_ER_BADSID	*SidList* contains a bad Sid. **RetSid* is set to the invalid Sid.
SEM_ER_BADSIDLIST	Bad *SidList*.
SEM_ER_CANCEL	Another user issued a SemCancel() call for one of the semaphores in *SidList*. The blocked SemAcquire() operation was cancelled, and no semaphores were acquired. **RetSid* is set to the Sid of the destroyed semaphore.
SEM_ER_CAPACITY_ASYNC_USER	SemSys async user table full
SEM_ER_CAPACITY_NODE	SemSys node table full.
SEM_ER_DESTROYED	Another user destroyed a semaphore that was being waited on by this user. The blocked acquire operation was cancelled. No semaphores were acquired. **RetSid* is set to the Sid of the destroyed semaphore.

SEM_ER_INTERRUPT	The blocked acquire operation was interrupted by an asynchronous event (such as a signal). The operation has been canceled.
SEM_ER_ISFROZEN	A *BlockOpt* of SEM_WAIT or SEM_TIMEOUT() was specified after the instance was frozen by the calling user.
SEM_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
SEM_ER_NOWAIT	*BlockOpt* of SEM_NOWAIT was specified and the request was not immediately satisfied.
SEM_ER_TIMEOUT	The time out period for the blocked acquire operation has expired without satisfying the request.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

semacquire *AcquireType SidList BlockingOpt*

ARGUMENTS

AcquireType **any**, **all** or **atomic**.

SidList List of Semaphore Ids separated by commas.

BlockingOpt See the Blocking Options discussion in the xipc command (Interactive Command Processor) section at the beginning of this Manual.

EXAMPLES

```
xipc> semacquire any 0,1,3 timeout(10)
      RetCode = 0   Sid = 1
```

4.2.4 SemCancel() - CANCEL BLOCKED OPERATIONS

NAME

`SemCancel()` - Cancel Blocked Operations

SYNTAX

```
#include "xipc.h"


XINT
SemCancel(SidList, RetSid)

SIDLIST SidList;
XINT *RetSid;
```

PARAMETERS

Name	Description
`SidList`	A list of Sids for which cancellation of blocked operations is to be performed. This list should be built by SemList() or SemListBuild() and updated by SemListAdd(). A pointer to a SIDLIST (type PSIDLIST) may be passed as well.
`RetSid`	A pointer to a variable that gets assigned by SemCancel() on return. It is acceptable to have a null RetSid argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired. For failed calls, where RC = SEM_ER_BADSID, **RetSid* is set to the invalid Sid. For successful calls **RetSid* is set to the last semaphore having its blocked operations cancelled. In all other cases, **RetSid* is undefined.

RETURNS

Value	Description
RC >= 0	Cancel successful. RC is the number of users having blocked operations cancelled.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemCancel() cancels blocked SemWait() or SemAcquire() operations involving the semaphores in *SidList*. Users blocked on any of the listed semaphores are notified of the cancellation of their SemWait() or SemAcquire() requests by receiving a `RC = SEM_ER_CANCEL`.

It is acceptable to have a null *RetSid* argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired.

ERRORS

Code	Description
SEM_ER_BADSID	*SidList* contains a bad Sid. **RetSid* is set to the invalid Sid.
SEM_ER_BADSIDLIST	Bad *SidList*.
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

semcancel *SidList*

ARGUMENTS

SidList List of Semaphore Ids separated by commas.

EXAMPLES

```
xipc> semcancel 0
      RetCode = 1   Sid = 0
```

4.2.5 SemClear() - CLEAR EVENT SEMAPHORES

NAME

`SemClear()` - Clear Event Semaphores

SYNTAX

```
#include "xipc.h"


XINT
SemClear(SidList, RetSid)

SIDLIST SidList;
XINT *RetSid;
```

PARAMETERS

Name	Description
`SidList`	A list of Sids to be cleared. This list should be built by SemList() or SemListBuild() and updated by SemListAdd(). A pointer to a SIDLIST (type PSIDLIST) may be passed as well.
`RetSid`	A pointer to a variable that gets assigned by SemClear() on return. It is acceptable to have a null RetSid argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired. For failed calls, where `RC = SEM_ER_BADSID` or `RC = SEM_ER_SEMCLEAR`, **RetSid* is set to the invalid Sid. For successful calls **RetSid* is set to the last semaphore cleared. In all other cases, **RetSid* is undefined.

RETURNS

Value	Description
RC >= 0	Clear successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemClear() clears the semaphores specified in *SidList*.
It is acceptable to have a null RetSid argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired.

ERRORS

Code	Description
`SEM_ER_BADSID`	*SidList* contains a bad Sid. **RetSid* is set to the invalid Sid.

SEM_ER_BADSIDLIST	Bad *SidList*.
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
SEM_ER_SEMCLEAR	*SidList* contains a Sid of a semaphore which is already clear. **RetSid* is set to that Sid.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

semclear *SidList*

ARGUMENTS

SidList List of Semaphore Ids separated by commas.

EXAMPLES

```
xipc> semclear 0,1
      RetCode = 0
```

4.2.6 SemCreate() - CREATE A NEW SEMAPHORE

NAME

`SemCreate()` - Create a New Semaphore

SYNTAX

```
#include "xipc.h"


XINT
SemCreate(Name, CreateValue)

CHAR *Name;
SEMVAL CreateValue;
```

PARAMETERS

Name	Description
`Name`	A pointer to a string that contains a symbolic name for publicly identifying the semaphore. *Name* must be null terminated and must not exceed `SEM_LEN_XIPCNAME` characters. If *Name* is `SEM_PRIVATE` then a private semaphore is created. Duplicate semaphore names (other than `SEM_PRIVATE`) are not permitted.
`CreateValue`	For resource semaphores, a non-zero positive integer representing the semaphore's maximum value. For event semaphores, either `SEM_CLEAR` or `SEM_SET` is specified.

RETURNS

Value	Description
RC >= 0	Create successful. RC is semaphore ID (Sid). Sid is to be used in all subsequent SemSys calls that refer to this semaphore.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemCreate() creates a new semaphore in SemSys. *Name* is used for publicly identifying the new semaphore. A *Name* of `SEM_PRIVATE` directs SemSys to create a private semaphore (i.e. having no public identification). A resource semaphore is created having an initial value of *CreateValue*, while an event semaphore is created either as *set* or as *clear*. The function returns the Sid of the created semaphore.

ERRORS

Code	Description
`SEM_ER_BADSEMNAME`	Invalid *Name* parameter.

SEM_ER_BADSEMVALUE	Invalid *CreateValue* parameter.
SEM_ER_CAPACITY_NODE	SemSys node table full.
SEM_ER_CAPACITY_TABLE	Semaphore table full.
SEM_ER_DUPLICATE	Semaphore with *Name* already exists.
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

semcreate {*Name* | @PRIVATE} *CreateValue*

ARGUMENTS

Name Name of new semaphore (or, if @PRIVATE, a private semaphore indicator).

CreateValue For a resource semaphore: A maximum value;
For an event semaphore: Either **clear** or **set**.

EXAMPLES

```
xipc> semcreate Resource 5
      Sid = 0

xipc> semcreate InitSem clear
      Sid = 3
```

4.2.7 SemDelete() - DELETE A SEMAPHORE

NAME

`SemDelete()` - Delete a Semaphore

SYNTAX

```
#include "xipc.h"


XINT
SemDelete(Sid)

XINT Sid;
```

PARAMETERS

Name	Description
Sid	The Semaphore ID of the semaphore to be deleted. *Sid* was obtained by the user via SemCreate() or SemAccess() function calls.

RETURNS

Value	Description
RC >= 0	Delete successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemDelete() deletes the semaphore identified by *Sid* from SemSys. SemDelete() will fail if any other users are holding onto or blocking for the specified semaphore.

ERRORS

Code	Description
SEM_ER_INVALIDSID	Invalid semaphore identifier specified.
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
SEM_ER_SEMBUSY	Semaphore *Sid* held or blocked on by other users.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

semdelete *Sid*

ARGUMENTS

Sid Semaphore Id.

EXAMPLES

```
xipc> semdelete 5
      RetCode = 0
```

4.2.8 SemDestroy() - DESTROY A SEMAPHORE

NAME

`SemDestroy()` - Destroy a Semaphore

SYNTAX

```
#include "xipc.h"


XINT
SemDestroy(Sid)

XINT Sid;
```

PARAMETERS

Name	Description
Sid	The semaphore ID of the semaphore to be destroyed. *Sid* was obtained by the user via SemCreate() or SemAccess() function calls.

RETURNS

Value	Description
RC >= 0	Destroy successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemDestroy() deletes the semaphore identified by *Sid* from SemSys even if other users are holding onto or blocked on the specified semaphore. Blocked SemAcquire() or SemWait() operations initiated by other users, having *Sid* as one of the semaphores being blocked on, are interrupted and returned with an `RC = SEM_ER_DESTROYED`, indicating the deletion of semaphore *Sid*. If *Sid* is a resource semaphore, its copies are silently taken away from users holding them. These users are not explicitly notified of the semaphore's deletion.

ERRORS

Code	Description
SEM_ER_INVALIDSID	Invalid semaphore identifier specified.
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

semdestroy *Sid*

ARGUMENTS

Sid Semaphore Id.

EXAMPLES

```
xipc> semdestroy 5
      RetCode = 0
```

4.2.9 SemFreeze() - FREEZE SEMSYS

NAME

`SemFreeze()` - Freeze SemSys

SYNTAX

```
#include "xipc.h"


XINT
SemFreeze()
```

PARAMETERS

None.

RETURNS

Value	Description
RC >= 0	SemFreeze successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemFreeze() freezes all SemSys activity occurring within the logged in instance, and gives the calling user exclusive access to all SemSys functionality. SemSys remains frozen until a SemUnfreeze(), XipcUnfreeze() or a XipcLogout() function call is issued.
SemFreeze() prevents all other users, working within the SemSys, from proceeding with SemSys operations - until a bracketing SemUnfreeze(), XipcUnfreeze() or XipcLogout() call is issued. The subsystem should therefore be kept frozen for as short a period of time as possible.

ERRORS

Code	Description
`SEM_ER_NOSUBSYSTEM`	SemSys is not configured in the instance.
`SEM_ER_NOTLOGGEDIN`	User not logged into instance (User never logged in, was aborted or disconnected).
`SEM_ER_ISFROZEN`	Calling user has already frozen SemSys.
`XIPCNET_ER_CONNECTLOST`	Connection to instance lost.
`XIPCNET_ER_NETERR`	Network transmission error.
`XIPCNET_ER_SYSERR`	Operating system error.

INTERACTIVE COMMAND

SYNTAX

semfreeze

ARGUMENTS

None.

EXAMPLES

```
xipc> semfreeze
      RetCode = 0
```

4.2.10 SemInfoSem() - GET SEMAPHORE INFORMATION

NAME

`SemInfoSem()` - Get Semaphore Information

SYNTAX

```
#include "xipc.h"


XINT
SemInfoSem(Sid, InfoSem)

XINT Sid;
SEMINFOSEM *InfoSem;
```

PARAMETERS

Name	Description
`Sid`	The Semaphore Id of the semaphore whose information is desired or `SEM_INFO_FIRST`, or `SEM_INFO_NEXT(`*`Sid`*`)`. *Sid* can be obtained via SemCreate() or SemAccess() function calls.
`InfoSem`	Pointer to a structure of type SEMINFOSEM, into which the semaphore information will be copied.

RETURNS

Value	Description
RC >= 0	Successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemInfoSem() fills the specified structure with information about the semaphore identified by *Sid*. The Sid argument can be specified as one of the following:

- *`Sid`* - a semaphore id identifying a specific semaphore
- `SEM_INFO_FIRST` - identifies the first valid semaphore id
- `SEM_INFO_NEXT(`*`Sid`*`)` - identifies the next valid semaphore id, following *`Sid`*.

A program reviewing the status of all semaphores within SemSys should call SemInfoSem() specifying `SEM_INFO_FIRST`, followed by repeated calls to the function specifying `SEM_INFO_NEXT` until the `SEM_ER_NOMORE` error code is returned.

Each SemSys semaphore has two lists of information associated with it:

- ☐ HList: The list of the Uids currently holding a copy of the specified resource semaphore. This HList is not used by event semaphores. The Uids are listed in the order that they acquired a semaphore copy.

- WList: For resource semaphores, the WList is the list of Uids currently waiting for a copy of the specified semaphore to become available. For event semaphores, the WList is the list of Uids currently waiting for the semaphore to be set. The Uids are listed in the order that they began waiting.

The SEMINFOSEM data structure follows:

```
/*
 * The SEMINFOSEM structure is used for retrieving status information
 * about a particular SemSys semaphore. SemInfoSem() fills the
 * structure with the data about the Sid it is passed.
 */

typedef struct _SEMINFOSEM
{
    XINT Sid;
    XINT CreateTime;                /* Time semaphore created */
    XINT CreateUid;                 /* The Uid who created it */
    XINT LastUid;                       /* Last Uid to use it */
    XINT MaxValue;                      /* Initial value */
    XINT CurValue;                      /* Current value */
    LBITS SemType;                      /* SEM_TYPE_RESOURCE or SEM_TYPE_EVENT */
    XINT HListTotalLength;
    XINT HListOffset;
    XINT HListLength;
    XINT WListTotalLength;
    XINT WListOffset;
    XINT WListLength;
    SEM_SEMHLISTITEM HList[SEM_LEN_INFOLIST];
    SEM_SEMWLISTITEM WList[SEM_LEN_INFOLIST];
    CHAR Name[SEM_LEN_XIPCNAME + 1]; /* Semaphore name */
}
SEMINFOSEM;
```

where:

HListTotalLength returns with the total internal length of the HList for this semaphore.

HListOffset is set by the user, prior to the SemInfoSem() function call, to specify the portion of the HList that should be returned (i.e. what offset to start from).

HListLength returns with the length of the HList portion returned by the current call to SemInfoSem(). More specifically, *HListLength* is the number of elements returned in the *HList* array. *HListLength* will be between 0 and `SEM_LEN_INFOLIST`.

HList is an array of list elements, where each element is of type `SEM_SEMHLISTITEM`. The `SEM_SEMHLISTITEM` data type is defined in `sempubd.h`.

The data structure follows; similar definitions and usage rules apply to the WList related fields.

```
typedef struct _SEM_SEMWLISTITEM
{
    XINT   Uid;
}
SEM_SEMWLISTITEM;
```

```
typedef struct _SEM_SEMHLISTITEM
{
    XINT   Uid;
}
SEM_SEMHLISTITEM;
```

A call to SemInfoSem() should be preceded by the setting of the *HListOffset* and *WListOffset* fields of the SEMINFOSEM structure to appropriate values.
For a full example and more complete information on using the Info functions, refer to the Advanced Topics chapter of the *X♦IPC* User Guide.

ERRORS

Code	Description
SEM_ER_INVALIDSID	Invalid semaphore identifier specified.
SEM_ER_BADLISTOFFSET	Invalid offset value specified.
SEM_ER_NOMORE	No more semaphores.
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

seminfosem *Sid* **| first | next(***Sid***) | all**

ARGUMENTS

Sid Print info on the **first** semaphore, the semaphore with Sid *Sid* or the **next** higher semaphore.

EXAMPLES

```
xipc> seminfosem 5
      Name: 'InitSem'  Type: Event
      Created by Uid: 22  At:...
      . . .
```

4.2.11 SemInfoSys() - GET SUBSYSTEM INFORMATION

NAME

SemInfoSys() - Get Subsystem Information

SYNTAX

```
#include "xipc.h"


XINT
SemInfoSys(InfoSys)

SEMINFOSYS *InfoSys;
```

PARAMETERS

Name	Description
InfoSys	Pointer to a structure of type SEMINFOSYS, into which the system information will be copied.

RETURNS

Value	Description
RC >= 0	Successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemInfoSys() fills the specified structure with information about the current instance of SemSys into which the user is logged in. The data structure follows:

```
/*
 * The SEMINFOSYS structure is used for retrieving status information
 * about the SemSys instance. SemInfoSys() fills the structure with the
 * data about the instance.
 */

typedef struct _SEMINFOSYS
{
    XINT MaxUsers;                      /* Maximum allowed users */
    XINT CurUsers;                      /* Current number of users */
    XINT MaxSems;                       /* Maximum allowed sems */
    XINT CurSems;                       /* Current number of sems */
    XINT MaxNodes;                      /* Max configured nodes */
    XINT FreeCnt;                       /* Current available nodes */
    CHAR Name[SEM_LEN_PATHNAME + 1]; /* InstanceFileName */
}
SEMINFOSYS;
```

For a full example and more complete information on using the Info functions, refer to the Advanced Topics chapter of the X♦IPC User Guide.

ERRORS

Code	Description
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

seminfosys

ARGUMENTS

None.

EXAMPLES

```
xipc> seminfosys
      Configuration:  '/usr/config'
      .......  Maximum  Current
      Users:        60       11
      .
      .
      .
```

4.2.12 SemInfoUser() - GET SEMSYS USER INFORMATION

NAME

SemInfoUser() - Get User Information

SYNTAX

```
#include "xipc.h"


XINT
SemInfoUser(Uid, InfoUser)

XINT Uid;
SEMINFOUSER *InfoUser;
```

PARAMETERS

Name	Description
Uid	The User Id of the user whose information is desired. or SEM_INFO_FIRST, or SEM_INFO_NEXT(*Uid*). *Uid* may be an asynchronous Uid (AUid).
InfoUser	Pointer to a structure of type SEMINFOUSER, into which the user information will be copied.

RETURNS

Value	Description
RC >= 0	Successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemInfoUser() fills the specified structure with information about the user identified by *Uid*. The Uid argument can be specified as one of the following:

- *Uid* - a user ID identifying a specific user
- SEM_INFO_FIRST - identifies the first valid user ID within the instance
- SEM_INFO_NEXT(*Uid*) - identifies the next valid user ID id, following *Uid*.

A program reviewing the status of all queues within SemSys should call SemInfoUser() specifying SEM_INFO_FIRST, followed by repeated calls to the function specifying SEM_INFO_NEXT until the SEM_ER_NOMORE error code is returned.

Each SemSys user has three lists of information associated with it:

- ☐ HList: The list of resource Sid copies currently held by the subject user. The Sids are listed in the order that they were acquired.
- ☐ QList: The list of Sids currently being requested by the subject user. The QList will have elements only when the user is blocked on a SemAcquire() or SemWait() operation.

- WList: The list of Sids currently being waited on by the subject user. The WList is the subset of the QList that has not yet been satisfied. It too will only have elements when the user is blocked on a SemAcquire() or SemWait() operation.

The SEMINFOUSER data structure follows:

```
/*
 * The SEMINFOUSER structure is used for retrieving status information
 * about a particular SemSys user. SemInfoUser() fills the structure
 * with the data about the Uid it is passed.
 */

typedef struct _SEMINFOUSER
{
    XINT Uid;
    XINT Pid;                           /* Process ID of user */
    TID  Tid;                           /* Thread ID of user */
    XINT LoginTime;                 /* Time of login to SemSys */
    XINT TimeOut;                       /* Remaining timeout secs */
    XINT WaitType;                      /* One of:  SEM_BLOCKEDATOMIC,
                                     * SEM_BLOCKEDALL, SEM_BLOCKEDANY or
                                         * SEM_USER_NOTWAITING
                                         */
    XINT HListTotalLength;
    XINT HListOffset;
    XINT HListLength;
    XINT WListTotalLength;
    XINT WListOffset;
    XINT WListLength;
    XINT QListTotalLength;
    XINT QListOffset;
    XINT QListLength;
    SEM_USERHLISTITEM HList[SEM_LEN_INFOLIST];
    SEM_USERWLISTITEM WList[SEM_LEN_INFOLIST];
    SEM_USERQLISTITEM QList[SEM_LEN_INFOLIST];
    CHAR Name[SEM_LEN_XIPCNAME + 1]; /* User login name */
    CHAR NetLoc[XIPC_LEN_NETLOC + 1];   /* Name of Client Node */
}
SEMINFOUSER;
```

where:

HListTotalLength returns with the total internal length of the HList for this user.

HListOffset is set by the user, prior to the SemInfoUser() function call, to specify the portion of the HList that should be returned (i.e. what offset to start from).

HListLength returns with the length of the HList portion returned by the current call to SemInfoUser(). More specifically, *HListLength* is the number of elements returned in the *HList* array. *HListLength* will be between 0 and SEM_LEN_INFOLIST.

HList is an array of list elements, where each element is of type SEM_USERHLISTITEM. The SEM_USERHLISTITEM data type is defined in sempubd.h.

Similar definitions and usage rules apply to the QList and WList related fields.

The data structures follow:

```
typedef struct _SEM_USERQLISTITEM
{
    XINT   Sid;
}
SEM_USERQLISTITEM;

typedef struct _SEM_USERWLISTITEM
{
    XINT   Sid;
}
SEM_USERWLISTITEM;

typedef struct _SEM_USERHLISTITEM
{
    XINT   Sid;
}
SEM_USERHLISTITEM;
```

A call to SemInfoUser() should be preceded by the setting of the *HListOffset*, *QListOffset* and *WListOffset* fields of the SEMINFOUSER structure to appropriate values.
For a full example and more complete information on using the Info functions, refer to the Advanced Topics chapter of the X•IPC User Guide.

ERRORS

Code	**Description**
SEM_ER_BADUID	Invalid *Uid* parameter.
SEM_ER_BADLISTOFFSET	Invalid offset value specified.
SEM_ER_NOMORE	No more users.
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

seminfouser *Uid* **| first | next(***Uid***) | all**

ARGUMENTS

Uid Print info on the **first** user, the user with Uid *Uid* or the **next** higher user.

EXAMPLES

```
xipc> seminfouser 9
      Name: 'MasterControl'  Pid: 214  Tid: 0
      Login Time:...
      . . .
```

4.2.13 SemList(), SemListBuild() - BUILD LISTS OF SIDS

NAME

SemList() - Create a One-Time List of Sids

SemListBuild() - Build a Reusable List of Sids

SYNTAX

```
#include "xipc.h"


PSIDLIST
SemList(Sid1, Sid2, ..., SEM_EOL)

SECTION Sid1;
SECTION Sid2;
...

PSIDLIST
SemListBuild(SidList, Sid1, Sid2, ..., SEM_EOL)

SIDLIST SidList;
SECTION Sid1;
SECTION Sid2;
...
```

PARAMETERS

Name	Description
SidList	An area to contain the resultant SIDLIST_xe "MIDLIST"_. A pointer to a SIDLIST (type PSIDLIST) may be passed as well.
Sid1, Sid2	Sids to be included in the resultant SIDLIST. SEM_EOL_xe "MEM_EOL"_ must be used to mark the end of the list.

RETURNS

Value	Description
RC != NULL	A pointer to the created list of Sids. For SemListBuild() it is a pointer to the *SidList* specified as an argument. For SemList() it is a pointer to an internal *SidList*.
RC == NULL	*SidList* exceeded SEM_LEN_SIDLIST elements.

DESCRIPTION

These functions are used for building lists of Sids in a format acceptable by SemSys functions taking a SIDLIST as one of their arguments. SEM_EOL must be the last argument to SemList() and SemListBuild().
SemListBuild() builds the list in the area specified by *SidList*. SemList() creates the list in an internal static area, and can therefore be safely used only once.

ERRORS

None.

4.2.14 SemListAdd(), SemListRemove() - UPDATE LIST OF SIDS

NAME

`SemListAdd()` – Add to a List of Sids

`SemListRemove()` – Remove from a List of Sids

SYNTAX

```
#include "xipc.h"

PSIDLIST
SemListAdd(SidList, Sid1, Sid2, ..., SEM_EOL)

SIDLIST SidList;
XINT Sid1;
XINT Sid2;
...

PSIDLIST
SemListRemove(SidList, Sid1, Sid2, ..., SEM_EOL)

SIDLIST SidList;
XINT Sid1;
XINT Sid2;
...
```

PARAMETERS

Name	Description
SidList	The SIDLIST to be updated_xe "QIDLIST"_. A pointer to a SIDLIST (type PSIDLIST) may be passed as well.
Sid1, Sid2,	The Sids to be added to or removed from the SIDLIST. SEM_EOL_xe "QUE_EOL"_ must be used to mark the end of the argument list.

RETURNS

Value	Description
RC != NULL	A pointer to the updated SIDLIST specified as an argument..
RC == NULL	The operation failed. The SIDLIST specified as an argument remains unchanged.

DESCRIPTION

These functions are used for modifying SIDLISTs by adding or removing Sidss. SEM_EOL must be the last argument to SemListAdd() and SemListRemove().

SemListAdd() adds Sids to an existing SIDLIST. The new Sids are added at the end of the specified SidList. If the number of Sids being added, plus the current number of Sids in the SIDLIST, exceeds SEM_LEN_SIDLIST, then the operation fails, NULL is returned and SidList remains unchanged.

SemListRemove() removes Sids from an existing SIDLIST. Each Sid must match a Sid of the SidList exactly, and then that Sid is removed. If it does not match a Sid of the SidList, the operation fails.

If the operation succeeds, a pointer to the modified argument SidList is returned; otherwise NULL is returned and the argument SidList remains unchanged.

ERRORS

None.

4.2.15 SemListCount() – GET NUMBER OF SIDS IN A LIST OF SIDS

NAME

`SemListCount()` - Get Number of Sids in a List of Sids

SYNTAX

```
#include "xipc.h"

XINT
SemListCount(SidList)

SIDLIST SidList;
```

PARAMETERS

Name	Description
`SidList`	A SIDLIST or a pointer to a SIDLIST (type PSIDLIST).

RETURNS

Value	Description
RC < 0	The SIDLIST is invalid.
RC > = 0	Number of Sids in SidList.

DESCRIPTION

SemListCount() is used for determining the number of Sids contained in SIDLIST.

ERRORS

None.

4.2.16 SemRelease() - RELEASE RESOURCE SEMAPHORES

NAME

SemRelease() - Release Resource Semaphores

SYNTAX

```
#include "xipc.h"


XINT
SemRelease(SidList, RetSid)

SIDLIST SidList;
XINT *RetSid;
```

PARAMETERS

Name	Description
SidList	A list of Sids being released. This list can be built by SemList() or SemListBuild() and updated by SemListAdd(). A pointer to a SIDLIST (type PSIDLIST) may be passed as well.
RetSid	A pointer to a variable that gets assigned by SemRelease() on return. It is acceptable to have a null RetSid argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired. For failed calls, where RC = SEM_ER_BADSID or RC = SEM_ER_SEMNOTHELD, **RetSid* is set to the invalid Sid. For successful calls **RetSid* is set to the last semaphore released. In all other cases, **RetSid* is undefined.

RETURNS

Value	Description
RC >= 0	Release successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemRelease() releases the resource semaphores specified in *SidList*.
It is acceptable to have a null *RetSid* argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired.

ERRORS

Code	Description
SEM_ER_BADSID	*SidList* contains a bad Sid. **RetSid* is set to the invalid Sid.

SEM_ER_BADSIDLIST	Bad *SidList*.
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
SEM_ER_SEMNOTHELD	*SidList* contains a Sid of a semaphore not currently held by the user. **RetSid* is set to that Sid.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

semrelease *SidList*

ARGUMENTS

SidList List of Semaphore Ids separated by commas.

EXAMPLES

```
xipc> semrelease 9,11
      RetCode = 0
```

4.2.17 SemSet() - SET EVENT SEMAPHORES

NAME

`SemSet()` - Set Event Semaphores

SYNTAX

```
#include "xipc.h"


XINT
SemSet(SidList, RetSid)

SIDLIST SidList;
XINT *RetSid;
```

PARAMETERS

Name	Description
`SidList`	A list of Sids to be set. This list should be built by SemList() or SemListBuild() and updated by SemListAdd(). A pointer to a SIDLIST (type PSIDLIST) may be passed as well.
`RetSid`	A pointer to a variable that gets assigned by SemSet() on return. It is acceptable to have a null RetSid argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired. For failed calls, where `RC = SEM_ER_BADSID` or `RC = SEM_ER_SEMSET`, **RetSid* is set to the invalid Sid. For successful calls **RetSid* is set to the last semaphore set. In all other cases, **RetSid* is undefined.

RETURNS

Value	Description
RC >= 0	Set successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemSet() sets the semaphores specified in *SidList*.
It is acceptable to have a null *RetSid* argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired.

ERRORS

Code	Description
`SEM_ER_BADSID`	*SidList* contains a bad Sid. **RetSid* is set to the invalid Sid.

SEM_ER_BADSIDLIST	Bad *SidList*.
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
SEM_ER_SEMSET	*SidList* contains a Sid of a semaphore which is already set. **RetSid* is set to that Sid.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

semset *SidList*

ARGUMENTS

SidList List of Semaphore Ids separated by commas.

EXAMPLES

```
xipc> semset 0,1,22
      RetCode = 0
```

4.2.18 SemUnfreeze() - UNFREEZE SEMSYS

NAME

`SemUnfreeze()` - Unfreeze SemSys

SYNTAX

```
#include "xipc.h"


XINT
SemUnfreeze()
```

PARAMETERS

None.

RETURNS

Value	Description
RC >= 0	SemUnfreeze successful.
RC < 0	Error (Error codes appear below.)

DESCRIPTION

SemUnfreeze() unfreezes SemSys. Other SemSys users are restored with equal access to the subsystem.
SemFreeze() prevents all other processes working within the SemSys, from proceeding with SemSys operations until a bracketing SemUnfreeze(), XipcUnfreeze() or XipcLogout() call is issued. The subsystems should therefore be kept frozen for as short a period of time as possible.
SemUnfreeze() will fail if the user has not previously frozen the SemSys via SemFreeze().

ERRORS

Code	Description
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
SEM_ER_NOTFROZEN	SemSys not frozen.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

semunfreeze

ARGUMENTS

None.

EXAMPLES

```
xipc> semunfreeze
      RetCode = 0
```

4.2.19 SemWait() - WAIT ON EVENT SEMAPHORES

NAME

SemWait() - Wait On Event Semaphores

SYNTAX

```
#include "xipc.h"

XINT
SemWait(WaitType, SidList, RetSid, Options)

XINT WaitType;
SIDLIST SidList;
XINT *RetSid;
... Options;
```

PARAMETERS

Name	Description
WaitType	SEM_ANY, SEM_ALL or SEM_ATOMIC depending on the waiting criteria desired; or SEM_USER_NOTWAITING
SidList	A list of Sids being waited on. This list should be a SIDLIST built by SemList() or SemListBuild() and updated by SemListAdd(). A pointer to a SIDLIST (type PSIDLIST) may be passed as well.
RetSid	A pointer to a variable that gets assigned by SemWait() upon its return. It is acceptable to have a null RetSid argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired. Successful wait operations (RC >= 0) return with **RetSid* equal to the last Sid to be set. Interrupted wait operations (RC = SEM_ER_DESTROYED or SEM_ER_CANCEL) return with **RetSid* equal to the destroyed or cancelled Sid. Failed calls with RC = SEM_ER_BADSID return with **RetSid* equal to the invalid Sid. **RetSid* is otherwise undefined.
Options	The *Options* parameter is of the form: [*ClearOpt* |] *BlockOpt* *ClearOpt* is optional and can be either SEM_CLEAR or SEM_NOCLEAR, indicating whether or not set semaphores are to be cleared at the completion of the SemWait() operation. SEM_NOCLEAR is the default *ClearOpt* value. See Appendix A, Using Blocking X•IPC Functions, for a description of *BlockOpt*.

RETURNS

Value	Description
RC >= 0	Wait successful. If *WaitType* = SEM_ANY then one of the requested semaphores has been set and **RetSid* is the Sid of that semaphore. If *WaitType* = SEM_ALL

or SEM_ATOMIC then all requested semaphores have been set and **RetSid* is the Sid of the last semaphore set.

RC < 0 Error (Error codes appear below.)

DESCRIPTION

SemWait() waits for the semaphores in *SidList* to be in a set state, based on the values of *WaitType* and *BlockOpt*.

The value of *WaitType* specifies how to satisfy the requested wait:

- If *WaitType* = SEM_ANY, the request is considered satisfied when any one of the semaphores in *SidList* is in the set state.
- If *WaitType* = SEM_ALL, the request is not considered satisfied until all the semaphores in *SidList* have been in the set state at least once since the beginning of the wait. Semaphore states are noted as they become set until the entire *SidList* has been set. If a semaphore that was noted to be set changes its state to clear before the wait is satisfied, the change will not affect the satisfaction of the wait.
- If *WaitType* = SEM_ATOMIC, the request is not considered satisfied until all the semaphores in *SidList* are in the set state at one time. Semaphore states are noted as they become set until the entire *SidList* has been set. If a semaphore that was noted to be set changes its state to clear before the wait is satisfied, the change will be noted and will prevent the wait from being satisfied. In this way it differs from *WaitType* = SEM_ALL.

The value of *ClearOpt* specifies whether semaphores should be left set or should be cleared once the Wait request has been completely satisfied:

- If *ClearOpt* = SEM_CLEAR, set semaphores are cleared when the request is considered satisfied.
- If *ClearOpt* = SEM_NOCLEAR or is omitted, set semaphores are left in their set state.

The value of *BlockOpt* specifies whether and how to block for the satisfaction of the requested wait in case it cannot be satisfied immediately. See Appendix A, Using Blocking *X♦IPC* Functions, for a description of how to use the blocking options.

A blocked call that is interrupted by an asynchronous signal or trap is returned with RC = SEM_ER_INTERRUPT.

Satisfied SemWait() calls return with **RetSid* equal to the Sid of the last semaphore to be set. It is acceptable to have a null *RetSid* argument; it is not necessary to declare and specify return variables for acquiring return values that are not desired.

ERRORS

Code	Description
SEM_ER_ASYNC	Operation is being performed asynchronously.
SEM_ER_ASYNCABORT	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
SEM_ER_BADBLOCKOPT	Invalid *BlockOpt*.
SEM_ER_BADCLEAROPT	Invalid *ClearOpt*.
SEM_ER_BADOPTION	Invalid *Option* parameter.

SEM_ER_BADSID	*SidList* contains a bad Sid. **RetSid* is set to the invalid Sid.
SEM_ER_BADSIDLIST	Bad *SidList*.
SEM_ER_BADWAITTYPE	Invalid *WaitType* parameter.
SEM_ER_CANCEL	Another user issued a SemCancel() call for one of the semaphores in *SidList*. The blocked SemWait() operation was cancelled. **RetSid* is set to the Sid of the semaphore for which the SemCancel() was issued.
SEM_ER_CAPACITY_ASYNC_USER	SemSys async user table full.
SEM_ER_CAPACITY_NODE	SemSys node table full.
SEM_ER_DESTROYED	Another user destroyed a semaphore that was being waited on by this user. The blocked acquire operation was cancelled. **RetSid* is set to the Sid of the destroyed semaphore.
SEM_ER_INTERRUPT	The blocked wait operation was interrupted by an asynchronous event (such as a signal). The operation has been canceled.
SEM_ER_ISFROZEN	A *BlockOpt* of SEM_WAIT or SEM_TIMEOUT() was specified after the instance was frozen by the calling user.
SEM_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.
SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
SEM_ER_NOWAIT	*BlockOpt* of SEM_NOWAIT was specified and the request was not immediately satisfied.
SEM_ER_TIMEOUT	The time out period for the blocked wait operation has expired without satisfying the request.
XIPCNET_ER_CONNECTLOST	Connection to instance lost.
XIPCNET_ER_NETERR	Network transmission error.
XIPCNET_ER_SYSERR	Operating system error.

INTERACTIVE COMMAND

SYNTAX

semwait *WaitType SidList [***Clear,***]BlockingOpt*

ARGUMENTS

WaitType **any**, **all** or **atomic**.

SidList List of Semaphore Ids separated by commas.

BlockingOpt See the Blocking Options discussion in the `xipc` command (Interactive Command Processor) section at the beginning of this Manual.

EXAMPLES

```
xipc> semwait any 0,1,3 clear,wait
      RetCode = 0   Sid = 1

xipc> semwait all 0,1 nowait
      RetCode = -1034
      XIPC NOWAIT
      Sid = 0
```

4.3 Macros

[There are no SemSys macros.]

5. APPENDICES

5.1 Appendix A: Using Blocking *X•IPC* Functions

5.1.1 BLOCKING OPTIONS

All *X•IPC* functions that have the potential to block or complete asynchronously, have a *BlockOpt* parameter that is used to specify the appropriate option for the function call.
The blocking option parameter accepts one of the following values, as listed in the table below. The characters "*XXX*_" in all blocking option codes and return codes should be replaced by "SEM_," "QUE_" or "MEM_", depending on the subsystem called.
The asynchronous options refer to a user-declared Asynchronous Result Control Block structure (ACB). The function of this control block is described in the next section.

Blocking Option	**Description**
*XXX*_NOWAIT	If the request specified in the function call cannot be satisfied, the function returns immediately with RC = *XXX*_ER_NOWAIT.
*XXX*_WAIT	If the request specified in the function call cannot be satisfied, the caller is blocked until the request is completed.
*XXX*_TIMEOUT(*n*)	If the request specified in the function call cannot be satisfied, the caller is blocked until the request is completed or until *n* seconds have elapsed after which the function returns with RC = *XXX*_ER_TIMEOUT.
*XXX*_CALLBACK(*Func, AcbPtr*)	The function returns immediately with RC = *XXX*_ER_ASYNC. When the request is completed, the ACB pointed to by *AcbPtr* is filled with the results of the operation, and the function *Func* is called with *AcbPtr* passed as its only argument.
*XXX*_POST(*Sid, AcbPtr*)	The function returns immediately with RC = *XXX*_ER_ASYNC. When the request is completed, the ACB pointed to by *AcbPtr* is filled with the results of the operation, and the event semaphore *Sid* is set.
*XXX*_IGNORE(*AcbPtr*)	The function returns immediately with RC = *XXX*_ER_ASYNC. When the request is completed, the ACB pointed to by *AcbPtr* is filled with the results of the operation.

The three asynchronous options, as described above, cause *all* successful operation completions to occur using the prescribed asynchronous mechanism - including operations that can be completed immediately.
It is important to note that flags are always ORed to the *left* of (before) the blocking option.

It is sometimes required that operations that complete immediately–without blocking–should return their result *synchronously* and have the specified asynchronous option apply *only* to blocking situations. This behavior can be achieved by specifying *XXX*_RETURN option flag along with the asynchronous options as in:

*XXX*_RETURN | *XXX*_CALLBACK(*Func, AcbPtr*)

*XXX*_RETURN | *XXX*_POST(*Sid, AcbPtr*)

*XXX*_RETURN | *XXX*_IGNORE(*AcbPtr*)

In each of the above cases the specified asynchronous mechanism is employed *only* if the operation cannot complete immediately. Operations that can complete immediately return synchronously with their results.

5.1.2 ASYNCHRONOUS RESULT CONTROL BLOCK (ACB)

The Asynchronous Result Control Block (ACB) is a data structure that is filled, upon completion of an asynchronous X•IPC operation, with the results of the operation. The results include the return code of the operation as well as operation dependent results (e.g., *RetQid* and *Priority*).
The ACB is declared by the caller of the asynchronous X•IPC operation and a pointer to it is specified in the *BlockOpt* parameter as described in the previous section. The ACB can be examined by the user to check the status of the asynchronous operation.
The pointer to the ACB structure is later passed as an argument to a user callback function, if the *XXX*_CALLBACK option had been specified as the *BlockOpt* parameter.

NOTE

The ACB must be allocated using the `static` storage attribute or by using dynamic allocation. It should not be allocated on the stack.

Example:

```
static ASYNCRESULT Acb;
```

Another Appendix contains the structure of the ACB.

5.1.3 CALLBACK ROUTINE

As we have seen, X•IPC provides the option for specifying a user-defined callback routine that is to be invoked when an asynchronous operation completes, i.e., the XXX_CALLBACK option. When the callback routine is invoked, the ACB associated with the operation already contains the results of the completed operation. A pointer to the ACB is passed as an argument to the callback routine.

The callback routine is defined as follows:

```
XINT
CallbackFunc(AcbPtr)
ASYNCRESULT *AcbPtr;
{
    .
    .
    .
}
```

5.2 Appendix B: Using Message Select Codes and Queue Select Codes

QueSys provides the systems developer with great flexibility in sending and receiving messages. It is this feature that most sets QueSys apart from existing message queuing facilities. The key to successful utilization of QueSys is a good understanding of when and how to use the various message and queue select codes. This section offers a brief tutorial that describes these 'whens and hows'.
All QueSys operations that dispatch or retrieve messages to and from QueSys queues require a *QueSelectCode* and a *QidList* argument. It is the combination of these two arguments that determines the destination queue of dispatched messages, as well as the identity of retrieved messages. It is therefore essential to understand the function of these two arguments and how they interact.
This document uses a shorthand notation for writing *QueSelectCode* and *QidList* argument specifications. Using this shorthand it is possible to examine and explore the open-ended possibilities afforded to the systems developer. Instead of formally describing the shorthand notation, the document demonstrates via examples.

5.2.1 DISPATCHING MESSAGES ONTO QUESYS QUEUES

Dispatching messages via QueSend() and QuePut() is presented first, since it is less complex than the retrieval of messages.
Dispatching messages onto QueSys message queues can be viewed as occurring in two steps:

- First, a list of one or more queues is defined.
- Then, the message is placed onto one of the queues in the list, depending on some criteria.

As an example, consider a programmer who wishes to send a message onto the shortest queue of the list of queues a, b and c (perhaps to guarantee balanced queue loads). The programmer would first define the queue list {a, b, c}, and then specify the 'Shortest Queue' criteria together with the queue list when dispatching the message using the QueSend() or QuePut() function calls. This can be easily expressed as:

```
QUE_Q_SHQ{a, b, c}
```

Similarly, the expression for sending a message onto the longest queue in the list would be:

```
QUE_Q_LNQ{a, b, c}
```

The syntax for such dispatch expressions is thus of the form:

```
QueSelectCode{QidList}
```

The different possible *QueSelectCode*s that may be used to dispatch a message using QuePut or QueSend are:

QUE_Q_SHQ	The shortest queue.
QUE_Q_LNQ	The longest queue.
QUE_Q_HPQ	The queue having the highest priority message.
QUE_Q_LPQ	The queue having the lowest priority message.
QUE_Q_EAQ	The queue having the earliest arrived (oldest) message.
QUE_Q_LAQ	The queue with the latest arrived (most recent) message.
QUE_Q_ANY	The first queue in the list that has room (not full).

Examples of their usage include:

`QUE_Q_LPQ{x, y, z}` Place the outgoing message on one of the queues x, y or z, having the lowest priority message.

`QUE_Q_EAQ{q, r, s}` Place the outgoing message on one of the queues q, r or s, having the earliest arrived (oldest) message. This selects queues in a 'least recently accessed' manner.

`QUE_Q_LAQ{m, n}` Place the outgoing message on one of the queues m or n, having the latest arrived (most recent) message.

`QUE_Q_SHQ{j, k, m}` Place the outgoing message on the shortest of the three queues j, k or m. This achieves queue balancing.

`QUE_Q_ANY{a, b, c}` Place the message on the first of the queue a, b or c that has room for another message. The queues are examined in the order of specification.

5.2.2 RETRIEVING MESSAGES FROM QUESYS QUEUES

Retrieving messages in the QueSys system can similarly be viewed as occurring in two steps, but with a minor variation:

- First a list of message queues is defined by the program. As part of this definition, one message is designated as the 'candidate message' for each of the listed queues, using a *MsgSelectCode*. For example, the specification

  `{QUE_M_HP(a), QUE_M_EA(b), QUE_M_LA(c)}`

  defines a list of three queues **a**, **b**, and **c**, where the candidate messages are:

  `QUE_M_HP(a),` the highest priority message on queue a.

  `QUE_M_EA(b),` the earliest arrived message on queue b.

  `QUE_M_LA(c),` the latest arrived message on queue c.

- A message then gets selected from the list of candidate messages using a *QueSelectCode*. The selected message is retrieved and returned to the calling function. Thus for example, the specification

  `QUE_Q_HP{QUE_M_EA(a), QUE_M_EA(b)}`

  compares the oldest (earliest arrived) messages on queue **a** and queue **b** and returns the one with the higher priority. Similarly, the specification

  `QUE_Q_EA{QUE_M_HP(x), QUE_M_HP(y), QUE_M_HP(z)}`

  returns the oldest of the highest priority messages from queues **x**, **y** and **z**.

  Now consider another retrieval example having a slightly different twist:

  `QUE_Q_LNQ{QUE_M_HP(a), QUE_M_HP(b), QUE_M_HP(c)}`

  The interpretation of this expression is as follows: First, the highest priority message on the three respective queues **a**, **b** and **c** are designated as candidate messages. The returned message is that candidate message which resides on the longest queue.

  Note that the `'QUE_Q_LNQ'` QueSelectCode when used in a candidate message selection capacity chooses the candidate message that resides on the longest queue of **a**, **b**, and **c**. This is a departure from the message retrieval examples demonstrated until now where the candidate message selection process was based on a 'QueSelectCode' that compared the designated candidate messages from each queue directly, one with the other. Here by contrast, the message selection is performed based on characteristics of the underlying queues.

The possible *MsgSelectCode*s are listed below.

QUE_M_EA(*Q*)	The earliest arrived (oldest) message on the queue *Q*.
QUE_M_LA(*Q*)	The latest arrived (most recent) message on the queue *Q*.
QUE_M_HP(*Q*)	The highest priority message on the queue *Q*.
QUE_M_LP(*Q*)	The lowest priority message on the queue *Q*.
QUE_M_PREQ(*Q, n*)	The first message on queue *Q* having a priority of *n*.
QUE_M_PRNE(*Q, n*)	The first message on queue *Q* *not* having a priority of *n*.
QUE_M_PRGT(*Q, n*)	The first message on queue *Q* with a priority greater than *n*.
QUE_M_PRGE(*Q, n*)	The first message on queue *Q* with a priority >= *n*.
QUE_M_PRLT(*Q, n*)	The first message on queue *Q* having a priority less than *n*.
QUE_M_PRLE(*Q, n*)	The first message on queue *Q* with a priority <= *n*.
QUE_M_PRRNG(*Q,n,m*)	The first message on queue *Q* with a priority in [*n*,*m*] range.
QUE_M_SEQEQ(q, seqn)	Designates the first message on queue q with a value equal to sequence number seqn.
QUE_M_SEQGE(q, seqn)	Designates the first message on queue q with a value greater than or equal to sequence number seqn.
QUE_M_SEQLE(q, seqn)	Designates the first message on queue q with a value less than or equal to sequence number seqn.
QUE_M_SEQGT(qid seqn)	Designates the first message on queue q with a value greater than sequence number seqn.
QUE_M_SEQLT(q, seqn)	Designates the first message on queue q with a value less than sequence number seqn.

Note that *MsgSelectCode*s involving priorities cause the queue to be searched in decreasing priority order.

The possible *QueSelectCode*s that can be used for selecting a candidate message from one of the listed queues during retrieval operations are the listed below. Beware of some of their differing interpretations as compared to their usage within message dispatch operations.

QUE_Q_EA	The earliest arrived (oldest) candidate message.
QUE_Q_LA	The latest arrived (most recent) candidate message.
QUE_Q_HP	The highest priority candidate message.
QUE_Q_LP	The lowest priority candidate message.
QUE_Q_LNQ	The candidate message from the longest queue in the list.
QUE_Q_SHQ	The candidate message from the shortest queue in the list.
QUE_Q_HPQ	The candidate message from the queue having the highest priority msg.
QUE_Q_LPQ	The candidate message from the queue having the lowest priority msg.
QUE_Q_EAQ	The candidate message from the queue having the earliest arrived msg.
QUE_Q_LAQ	The candidate message from the queue having the latest arrived msg.
QUE_Q_ANY	The first candidate message.

5.2.3 EXPRESSION SIMPLIFICATION

Expression simplification can be employed in certain cases. Simplification is straight forward, involving simple defaults.
Whenever a message retrieval *QidList* has an entry wherein which a MsgSelectCode is not provided for a given queue (i.e., only the Qid is given), then the retrieval operation's QueSelectCode is employed as the message select criteria for that given queue.
The following examples demonstrate this concept. The following two message retrieval expressions are equivalent:

```
QUE_Q_HP{QUE_M_HP(x), QUE_M_EA(y), QUE_M_HP(z)}
QUE_Q_HP{x, QUE_M_EA(y), z}
```

They both consider three candidate messages:

The highest priority message on queue **x**.
The earliest arrived message on queue **y**.
The highest priority message on queue **z**.

The candidate message having the highest priority is the one retrieved.
Note that the first and third Qids of the simplified expression lack a *MsgSelectCode*. As a result they inherit the criteria of the expression's *QueSelectCode* (Highest Priority).
Similarly:

```
QUE_Q_HP{QUE_M_HP(q), QUE_M_HP(r), QUE_M_HP(s)}
QUE_Q_HP{q, r, s}
```

Both of these retrieval expressions return the overall highest priority message found on the three queues **q**, **r** and **s**.
How the expression *QUE_Q_HP{q, r, s}* returns the highest priority message of all three queues **q**, **r** and **s** is accomplished as follows (considering the unsimplified form of the expression):

```
QUE_Q_HP{QUE_M_HP(q), QUE_M_HP(r), QUE_M_HP(s)}
```

First, the candidate messages from the three queues **q**, **r** and **s** are designated. They are the highest priority message of their respective queues. These three candidate messages are then compared and the highest priority message of the three candidates is chosen.
Note, therefore, that a *QidList* of the form *{q, r, s}* can be used interchangeably within message dispatch and retrieval functions.

5.2.4 PRIORITY SPECIFICATION DURING RETRIEVAL

A number of the *MsgSelectCode*s deal with priorities. A variety of priority values or ranges can be specified.
For example:

```
QUE_Q_EA{QUE_M_PREQ(a, 100), QUE_M_PRLT(b, 50)}
```

designates the first message on queue **a** having a priority of 100 as the candidate message of queue **a**, and the first message on queue **b** having a priority less than 50 as the candidate message of queue **b**. It then returns the earliest arrived (oldest) of these two candidate messages.
Similarly:

```
QUE_Q_LNQ{QUE_M_PRRNG(a, 100, 200), QUE_M_PRRNG(b, 100, 200)}
```

considers the first message on queue **a** having a priority in the range [100,200], and does the same for queue **b**. It then returns the candidate message from the longer of the two queues.

5.2.5 CONCLUSION

This tutorial has outlined a few guidelines and examples of how to dispatch and retrieve messages to and from queues within the *X•IPC* QueSys subsystem. The possible combinations are far more numerous than can be presented in a manual. These examples and the shorthand used to express them should provide a good starting point for using the system correctly and to its full potential.

5.3 Appendix C: X•IPC User Data Structures

5.3.1 X•IPC GENERAL DATA STRUCTURES

NAME

X•IPC **General Data Structures** - Data Structures Used by all X•IPC subsystems

SYNTAX

```
/*
 *  The ASYNCRESULT Control Block (ACB) structure is used for examining
 *  the results of an asynchronous operation.  The structure contains
 *  a union that defines returned fields for every XIPC operation
 *  that may block.
 */

/*********************************************************************
**   Macros
*********************************************************************/

#define XIPC_ASYNC_INPROGRESS   1
#define XIPC_ASYNC_COMPLETED    2

#define ACB_FIELD(AcbPtr, Function, Field)  AcbPtr->Api.Function.Field

/*********************************************************************
**  'ACB' - ASYNCRESULT Control Block ---
*********************************************************************/

struct _ASYNCRESULT         /* Result of Async API call */
{
   XINT  AUid;          /* Async Uid  "receipt" */
   XINT  AsyncStatus;       /* XIPC_ASYNC_INPROGRESS
                               or XIPC_ASYNC_COMPLETED */
   XINT  UserData1;     /* ------ user defined usage ---- */
   XINT  UserData2;     /* ------ user defined usage ---- */
   XINT  UserData3;     /* ------ user defined usage ---- */

   XINT      OpCode;       /* Async operation, key to union */

   union
   {
       struct
       {
           XINT        RetSid;
           XINT        RetCode;/* of completed async operation */
       }
       SemWait;

       struct
       {
           XINT        RetSid;
           XINT        RetCode;/* of completed async operation */
       }
       SemAcquire;
```

```
struct
{
    MSGHDR      MsgHdr; /* The resultant MsgHdr */
    CHAR FAR    *MsgBuf;
    XINT        RetCode;/* of completed async operation */
}
QueWrite;

struct
{
    MSGHDR      MsgHdr; /* The resultant MsgHdr */
    XINT        RetQid;
    XINT        RetCode;
}
QuePut;

struct
{
    MSGHDR      MsgHdr; /* The resultant MsgHdr */
    XINT        Priority;
    XINT        RetQid;
    XINT        RetCode;
}
QueGet;

struct
{
    CHAR FAR    *MsgBuf;
    XINT        RetQid;
    XINT        RetCode;
}
QueSend;

struct
{
    CHAR FAR    *MsgBuf;
    XINT        MsgLen;
    XINT        Priority;
    XINT        RetQid;
    XINT        RetCode;
}
QueReceive;

struct
{
    /*
     * Only used for passing error info re
     * failed QueBurstSend() operation.
     */

    XINT        SeqNo;      /* of burst-send message that failed */
    XINT        TargetQid;
    XINT        Priority;
    XINT        RetQid;
    XINT        RetCode;
}
QueBurstSend;
```

```
struct
{
    /*
     * Only used for handling an asynchronous
     * QueBurstSendSync() operation.
     */
    XINT        SeqNo;      /* of last burst-send msg enqueued */
    XINT        RetCode;
}
QueBurstSendSync;
```

```
struct
{
    XINT        Mid;        /* of target */
    XINT        Offset; /* of target */
    XINT        Length; /* of target */
    CHAR FAR    *Buffer;
    XINT        RetCode;
}
MemWrite;

struct
{
    XINT        Mid;        /* of target */
    XINT        Offset; /* of target */
    XINT        Length; /* of target */
    CHAR FAR    *Buffer;
    XINT        RetCode;
}
MemRead;

struct
{
    SECTION     RetSec;
    XINT        RetCode;
}
MemSecOwn;

struct
{
    SECTION     RetSec;
    XINT        RetCode;
}
MemLock;

struct
{
    MOM_MSGID   MsgId;
    XINT        RetCode;
}
MomSend;

struct
{
    CHAR FAR    *MsgBuf;
    XINT        MsgLen;
    MOM_MSGID   MsgId;
    XINT        ReplyAppQueue;
    XINT        RetCode;
    XINT        TrackingLevel;
}
MomReceive;

struct
{
    XINT        RetCode;
}
MomEvent;
```

```
    }
    Api;

};
```

5.3.2 QUESYS DATA STRUCTURES

NAME

QueSys Data Structures - Data Structures Used Within QueSys

SYNTAX

```
/*
 * The MSGHDR structure is used for manipulating QueSys message
 * headers. Each active message in an instance has a message header
 * associated with it.
 */

typedef struct _MSGHDR
{
    XINT GetQid;                    /* Last Qid msg was on */
    XINT HdrStatus;                     /* Rmvd or Not Rmvd, etc */
    XINT Priority;                      /* Message's priority */
    XINT SeqNum;                    /* Msg sequence # within queue */
    XINT TimeVal;                       /* Msg sequence number within QueSys */
    XINT Size;                          /* Numb. of bytes in msg */
    XINT TextOffset;                /* Offset of msg's text in text-pool */
    XINT Uid;                           /* The User-Id of user that sent msg */
    CHAR Data[MSGHDR_DATASIZE];         /* User data field */
}
MSGHDR;
```

```
/*
 * The QUEINFOQUE structure is used for retrieving status information
 * about a particular QueSys message queue. QueInfoQue() fills the
 * structure with the data about the Qid it is passed.
 */
typedef struct _QUEINFOQUE
{
    XINT Qid;
    XINT CreateTime;                /* Time queue was created */
    XINT CreateUid;                 /* The Uid who created it */
    XINT LastUid;                       /* Last Uid to use queue */
    LBITS QueType;                      /*  - Not Used - */
    XINT LimitMessages;             /* Max message capacity */
    XINT LimitBytes;                /* Max byte capacity */
    XINT CountMessages;             /* Current number of msgs */
    XINT CountBytes;                /* Current number of bytes */
    XINT CountIn;                       /* Number msgs entered que */
    XINT CountOut;                      /* Number msgs exited que */
    XINT LastUidGet;                /* Last Uid to put a msg */
    XINT LastUidPut;                /* Last Uid to get a msg */
    XINT SpoolFlag;                 /* Spooling: ON or OFF */
    XINT SpoolMessages;             /* Number msgs spooled */
    XINT SpoolBytes;                /* Number bytes spooled */
    XINT WListTotalLength;
    XINT WListOffset;
    XINT WListLength;
    QUE_QUEWLISTITEM WList[QUE_LEN_INFOLIST];
    CHAR SpoolFileName[QUE_LEN_PATHNAME+1];
    CHAR Name[QUE_LEN_XIPCNAME + 1]; /* Queue name */
}
QUEINFOQUE;
```

```
typedef struct _QUE_QUEWLISTITEM
{
    XINT  OpCode;                       /* PUT or GET */
    union
    {
        struct
        {
            XINT Uid;                   /* User blocked */
            XINT MsgSize;               /* Putting Msg */
            XINT MsgPrio;               /* Msg Priority */
        }
        Put;

        struct
        {
            XINT Uid;                   /* User blocked */
            XINT MsgSelCode;        /* Getting Msg */
            XINT Parm1;
            XINT Parm2;
        }
        Get;
    }
    u;
}
QUE_QUEWLISTITEM;

/*
 * The QUEINFOUSER structure is used for retrieving status information
 * about a particular QueSys user. QueInfoUser() fills the structure
 * with the data about the Uid it is passed.
 */

typedef struct _QUEINFOUSER
{
    XINT Uid;
    XINT Pid;                           /* Process ID of user */
    TID  Tid;                           /* Thread ID of user */
    XINT LoginTime;                 /* Time of login to QueSys */
    XINT TimeOut;                       /* Remaining timeout secs */
    XINT WaitType;                      /* One of:  QUE_BLOCKEDWRITE,
QUE_BLOCKEDPUT,                          * QUE_BLOCKEDGET or QUE_USER_NOTWAITING
                                         */
    XINT CountPut;                      /* Number of msgs put */
    XINT CountGet;                      /* Number of msgs gotten */
    XINT LastQidPut;                /* Last Qid msg was put on */
    XINT LastQidGet;                /* Last Qid msg taken from */
    XINT WListTotalLength;
    XINT WListOffset;
    XINT WListLength;
    QUE_USERWLISTITEM WList[QUE_LEN_INFOLIST];
    CHAR Name[QUE_LEN_XIPCNAME + 1]; /* User login name */
    CHAR NetLoc[XIPC_LEN_NETLOC + 1];    /* Name of Client Node */
}
QUEINFOUSER;

typedef struct _QUE_USERWLISTITEM
{
    XINT  OpCode;                       /* PUT, GET or WRITE */
```

```
    union
    {
        struct
        {
            XINT Qid;                   /* Que Blocked */
            XINT MsgSize;               /* Putting Msg */
            XINT MsgPrio;               /* Msg Priority */
        }
        Put;

        struct
        {
            XINT Qid;                   /* Que Blocked */
            XINT MsgSelCode;        /* Getting Msg */
            XINT Parm1;
            XINT Parm2;
        }
        Get;

        struct
        {
            XINT MsgSize;               /* Write Blked */
        }
        Write;
    }
    u;
}
QUE_USERWLISTITEM;
```

```
/*
 * The QUEINFOSYS structure is used for retrieving status information
 * about the QueSys instance. QueInfoSys() fills the structure with the
 * data about the instance.
 */

typedef struct _QUEINFOSYS              /* system information */
{
    XINT MaxUsers;                      /* Max configured users */
    XINT CurUsers;                      /* Number of current users */
    XINT MaxQueues;                 /* Max configured queues */
    XINT CurQueues;                 /* Number of current queues */
    XINT MaxNodes;                      /* Max configured nodes */
    XINT FreeNCnt;                      /* Current available nodes */
    XINT MaxHeaders;                /* Max configured headers */
    XINT FreeHCnt;                      /* Current available hdrs */
    XINT SplTickSizeBytes;              /* Configured spool tick value */
    XINT MsgPoolSizeBytes;              /* Configured text pool size */
    XINT MsgTickSize;                   /* Configured text tick size */
    XINT MsgPoolTotalAvail;             /* Free text pool space */
    XINT MsgPoolLargestBlk;             /* Largest contig block */
    XINT MsgPoolMaxPosBlks;             /* Max possible tick blocks */
    XINT MsgPoolTotalBlks;              /* Number allocated blocks */
    XINT WListTotalLength;
    XINT WListOffset;
    XINT WListLength;
    QUE_SYSWLISTITEM WList[QUE_LEN_INFOLIST];
    CHAR Name[QUE_LEN_PATHNAME + 1]; /* InstanceFileName */
}
QUEINFOSYS;

typedef struct _QUE_SYSWLISTITEM
{

    XINT Uid;                           /* User Blked */
    XINT MsgSize;                       /* Write size */
}
QUE_SYSWLISTITEM;
```

5.3.3 MEMSYS DATA STRUCTURES

NAME

MemSys Data Structures - Data Structures Used Within MemSys

SYNTAX

```
/*
 * The SECTION structure is used for manipulating MemSys section
 * overlays. MemSection() can be used to initialize a section with
 * values.
 */

typedef struct _SECTION
{
    XINT   Mid;          /* MemSys memory seg ID   */
    XINT   Offset;           /* Offset into segment     */
    XINT   Size;         /* Byte size of section   */
}
SECTION;


/*
 * The MEMINFOSEC structure is used for retrieving status information
 * about a particular MemSys section overlay. MemInfoSec() fills the
 * structure with the data about the Section it is passed.
 */

typedef struct _MEMINFOSEC
{
    XINT Mid;                /* MemSys segment ID */
    XINT Offset;         /* Offset into the segment */
    XINT Size;               /* Section size in bytes */
    XINT OwnerUid;           /* Uid of section owner */
    XINT OwnerPriv;          /* Owner access privileges */
    XINT OtherPriv;          /* Other access privileges */
}
MEMINFOSEC;
```

```
/*
 * The MEMINFOMEM structure is used for retrieving status information
 * about a particular MemSys semaphore. MemInfoMem() fills the
 * structure with the data about the Mid it is passed.
 */

typedef struct _MEMINFOMEM
{
    XINT Mid;
    XINT CreateTime;                /* Time segment was created */
    XINT CreateUid;                 /* The Uid who created it */
    XINT Size;                          /* Size of segment (bytes)*/
    XINT NumSections;                   /* Num of sections on seg */
    XINT NumSecOwned;                   /* Num of owned sections  */
    XINT NumSecLocked;                  /* Num of locked sections */
    XINT NumBytesOwned;             /* Bytes owned on segment */
    XINT NumBytesLocked;            /* Bytes locked on segment */
    XINT CountWrite;                /* Num writes to segment  */
    XINT CountRead;                 /* Num reads from segment */
    XINT LastUidWrite;                  /* Last Uid to write segment */
    XINT LastUidRead;                   /* Last Uid to read segment */
    XINT LastUidOwned;                  /* Last Uid to own on segment */
    XINT LastUidLocked;             /* Last Uid to lock on segment */
    XINT SListTotalLength;
    XINT SListOffset;
    XINT SListLength;
    XINT WListTotalLength;
    XINT WListOffset;
    XINT WListLength;
    MEM_MEMSLISTITEM SList[MEM_LEN_INFOLIST];
    MEM_MEMWLISTITEM WList[MEM_LEN_INFOLIST];
    CHAR Name[MEM_LEN_XIPCNAME + 1]; /* Segment name */
}
MEMINFOMEM;

typedef struct _MEM_MEMSLISTITEM
{
    XINT OwnerUid;
    XINT OwnerPriv;
    XINT OtherPriv;
    XINT Mid;
    XINT Offset;
    XINT Size;
}
MEM_MEMSLISTITEM;

typedef struct _MEM_MEMWLISTITEM
{
    XINT Uid;
    XINT OpCode;                /* MEM_BLOCKEDLOCK, MEM_BLOCKEDREAD,
                                    * MEM_BLOCKEDWRITE or MEM_BLOCKEDOWN
                                    */
    XINT Offset;
    XINT Size;
}
MEM_MEMWLISTITEM;
```

```
/*
 * The MEMINFOUSER structure is used for retrieving status information
 * about a particular MemSys user. MemInfoUser() fills the structure
 * with the data about the Uid it is passed.
 */

typedef struct _MEMINFOUSER
{
    XINT Uid;
    XINT Pid;                           /* Process Id of user  */
    TID  Tid;                           /* Thread ID of user  */
    XINT LoginTime;                 /* Time of login to MemSys */
    XINT TimeOut;                       /* Remaining timeout secs */
    XINT WaitType;                      /* One of: MEM_BLOCKEDWRITE,
                                         * MEM_BLOCKEDREAD, MEM_BLOCKEDOWN,
                                         * MEM_BLOCKEDLOCK or MEM_USER_NOTWAITING
                                         */
    XINT NumSecOwned;                   /* Num sects owned by Uid */
    XINT NumSecLocked;                  /* Num sects locked by Uid */
    XINT NumBytesOwned;             /* Num bytes owned by Uid */
    XINT NumBytesLocked;            /* Num bytes locked by Uid */
    XINT CountWrite;                /* Num of Uid write opers */
    XINT CountRead;                 /* Num of Uid read opers  */
    XINT HListTotalLength;
    XINT HListOffset;
    XINT HListLength;
    XINT WListTotalLength;
    XINT WListOffset;
    XINT WListLength;
    XINT QListTotalLength;
    XINT QListOffset;
    XINT QListLength;
    MEM_USERHLISTITEM HList[MEM_LEN_INFOLIST];
    MEM_USERWLISTITEM WList[MEM_LEN_INFOLIST];
    MEM_USERQLISTITEM QList[MEM_LEN_INFOLIST];
    CHAR Name[MEM_LEN_XIPCNAME + 1]; /* User login name */
    CHAR NetLoc[XIPC_LEN_NETLOC + 1];   /* Name of Client Node */
}
MEMINFOUSER;

typedef struct _MEM_USERQLISTITEM
{
    XINT Mid;
    XINT Offset;
    XINT Size;
}
MEM_USERQLISTITEM;

typedef struct _MEM_USERHLISTITEM
{
    XINT OpCode;                    /* MEM_BLOCKEDLOCK or MEM_BLOCKEDOWN */
    XINT Mid;
    XINT Offset;
    XINT Size;
}
MEM_USERHLISTITEM;
```

```
typedef struct _MEM_USERWLISTITEM
{
    XINT Mid;
    XINT Offset;
    XINT Size;
}
MEM_USERWLISTITEM;

/*
 * The MEMINFOSYS structure is used for retrieving status information
 * about the MemSys instance. MemInfoSys() fills the structure with the
 * data about the instance.
 */

typedef struct _MEMINFOSYS
{
    XINT MaxUsers;                      /* Max configured users */
    XINT CurUsers;                      /* Current num of users */
    XINT MaxSegments;                   /* Max configured segments */
    XINT CurSegments;                   /* Current num of segments */
    XINT MaxNodes;                      /* Max configured nodes */
    XINT FreeNCnt;                      /* Current available nodes */
    XINT MaxSections;                   /* Max configured sections */
    XINT FreeSCnt;                      /* Current available sects */
    XINT MemPoolSizeBytes;              /* Configured mem pool size */
    XINT MemTickSize;                   /* Configured mem tick size */
    XINT MemPoolTotalAvail;          /* Free text pool space */
    XINT MemPoolLargestBlk;          /* Largest contig block */
    XINT MemPoolMaxPosBlks;          /* Max possible tick blocks */
    XINT MemPoolTotalBlks;               /* Number allocated blocks */
    CHAR Name[MEM_LEN_PATHNAME + 1]; /* InstanceFilename */
}
MEMINFOSYS;
```

5.3.4 SEMSYS DATA STRUCTURES

NAME

SemSys Data Structures - Data Structures Used Within SemSys

SYNTAX

```
/*
 * The SEMINFOSEM structure is used for retrieving status information
 * about a particular SemSys semaphore. SemInfoSem() fills the
 * structure with the data about the Sid it is passed.
 */

typedef struct _SEMINFOSEM
{
    XINT Sid;
    XINT CreateTime;                /* Time semaphore created */
    XINT CreateUid;                 /* The Uid who created it */
    XINT LastUid;                       /* Last Uid to use it */
    XINT MaxValue;                      /* Initial value */
    XINT CurValue;                      /* Current value */
    LBITS SemType;                      /* SEM_TYPE_RESOURCE or SEM_TYPE_EVENT */
    XINT HListTotalLength;
    XINT HListOffset;
    XINT HListLength;
    XINT WListTotalLength;
    XINT WListOffset;
    XINT WListLength;
    SEM_SEMHLISTITEM HList[SEM_LEN_INFOLIST];
    SEM_SEMWLISTITEM WList[SEM_LEN_INFOLIST];
    CHAR Name[SEM_LEN_XIPCNAME + 1]; /* Semaphore name */
}
SEMINFOSEM;


typedef struct _SEM_SEMWLISTITEM
{
    XINT   Uid;
}
SEM_SEMWLISTITEM;


typedef struct _SEM_SEMHLISTITEM
{
    XINT   Uid;
}
SEM_SEMHLISTITEM;
```

```
/*
 * The SEMINFOUSER structure is used for retrieving status information
 * about a particular SemSys user. SemInfoUser() fills the structure
 * with the data about the Uid it is passed.
 */

typedef struct _SEMINFOUSER
{
    XINT Uid;
    XINT Pid;                           /* Process ID of user */
    TID  Tid;                           /* Thread ID of user */
    XINT LoginTime;                 /* Time of login to SemSys */
    XINT TimeOut;                       /* Remaining timeout secs */
    XINT WaitType;                      /* One of:  SEM_BLOCKEDATOMIC,
                                     * SEM_BLOCKEDALL, SEM_BLOCKEDANY or
                                         * SEM_USER_NOTWAITING
                                         */
    XINT HListTotalLength;
    XINT HListOffset;
    XINT HListLength;
    XINT WListTotalLength;
    XINT WListOffset;
    XINT WListLength;
    XINT QListTotalLength;
    XINT QListOffset;
    XINT QListLength;
    SEM_USERHLISTITEM HList[SEM_LEN_INFOLIST];
    SEM_USERWLISTITEM WList[SEM_LEN_INFOLIST];
    SEM_USERQLISTITEM QList[SEM_LEN_INFOLIST];
    CHAR Name[SEM_LEN_XIPCNAME + 1]; /* User login name */
    CHAR NetLoc[XIPC_LEN_NETLOC + 1];   /* Name of Client Node */
}
SEMINFOUSER;

typedef struct _SEM_USERQLISTITEM
{
    XINT   Sid;
}
SEM_USERQLISTITEM;

typedef struct _SEM_USERWLISTITEM
{
    XINT   Sid;
}
SEM_USERWLISTITEM;
```

```
typedef struct _SEM_USERHLISTITEM
{
    XINT   Sid;
}
SEM_USERHLISTITEM;

/*
 * The SEMINFOSYS structure is used for retrieving status information
 * about the SemSys instance. SemInfoSys() fills the structure with the
 * data about the instance.
 */

typedef struct _SEMINFOSYS
{
    XINT MaxUsers;                      /* Maximum allowed users */
    XINT CurUsers;                      /* Current number of users */
    XINT MaxSems;                       /* Maximum allowed sems */
    XINT CurSems;                       /* Current number of sems */
    XINT MaxNodes;                      /* Max configured nodes */
    XINT FreeCnt;                       /* Current available nodes */
    CHAR Name[SEM_LEN_PATHNAME + 1]; /* InstanceFileName */
}
SEMINFOSYS;
```

5.4 Appendix D: QueSys/SemSys/MemSys Error Codes

5.4.1 QUESYS ERROR CODES: BY SYMBOLIC ERROR NAME

Symbolic Error Name	Number	Description
QUE_ER_ASYNC	-1097	Operation is being performed asynchronously.
QUE_ER_ASYNCABORT	-1098	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
QUE_ER_BADBLOCKOPT	-1031	Invalid *BlockOpt*.
QUE_ER_BADBUFFER	-1019	*MsgBuf* is NULL.
QUE_ER_BADDIRECTION	-1620	Invalid *Direction* parameter.
QUE_ER_BADERROROPT	-1667	Invalid *ErrorOption* parameter.
QUE_ER_BADFILENAME	-1022	Invalid *SpoolFileName* specified.
QUE_ER_BADLENGTH	-1614	Invalid *MsgLength* parameter.
QUE_ER_BADLIMIT	-1615	Invalid *LimitMsgs* or *LimitBytes* parameter.
QUE_ER_BADLISTOFFSET	-1014	Invalid offset value specified.
QUE_ER_BADMSGSELECTCODE	-1618	Invalid MsgSelectCode within *QidList*.
QUE_ER_BADOPTION	-1030	Invalid *Options* parameter.
QUE_ER_BADPRIORITY	-1616	Invalid *Priority* parameter.
QUE_ER_BADQID	-1612	Bad *TargetQid*, or QUE_NULL_QID was specified when valid Qid is required.
QUE_ER_BADQIDLIST	-1613	Invalid QidList parameter.
QUE_ER_BADQUENAME	-1611	Invalid *Name* parameter.
QUE_ER_BADQUESELECTCODE	-1619	Invalid QueSelectCode parameter.
QUE_ER_BADREADAHEAD	-1671	Invalid ReadAheadBufSize parameter.
QUE_ER_BADSID	-1610	*Sid* is not a valid semaphore ID.
QUE_ER_BADSYNCMODE	-1674	Invalid *Mode* parameter.
QUE_ER_BADTEXT	-1617	*MsgHdr* has invalid text pointer.
QUE_ER_BADTRIGGERCODE	-1051	Bad trigger code.
QUE_ER_BADUID	-1023	No user with specified *Uid*.
QUE_ER_BADVAL	-1024	Illegal trigger parameter value.
QUE_ER_CAPACITY_ASYNC_USER	-1645	QueSys async user table full.
QUE_ER_CAPACITY_HEADER	-1642	QueSys header table full.
QUE_ER_CAPACITY_NODE	-1643	QueSys node table full.
QUE_ER_CAPACITY_TABLE	-1644	Queue table full.
QUE_ER_CAPACITY_USER	-1641	QueSys user table full.

Symbolic Error Name	Number	Description
QUE_ER_DESTROYED	-1035	Another user destroyed a queue that a blocked QueBurstSend() call was waiting to enqueue onto. The blocked QueBurstSend() operation was canceled. No message was enqueued.
QUE_ER_DUPLICATE	-1032	Queue with *Name* already exists.
QUE_ER_ENDOFQUEUE	-1625	An end of the queue has been reached.
QUE_ER_FAILSTART	-1651	QueSys initialization failed.
QUE_ER_FAILSTOP	-1652	QueSys termination failed.
QUE_ER_GHOSTSTART	-1653	Cannot register QueSys with *X•IPC* object daemon.
QUE_ER_GHOSTSTOP	-1654	Cannot deregister QueSyswith *X•IPC* object daemon.
QUE_ER_INRECEIVEBURST	-1666	User is in a receive-burst.
QUE_ER_INSENDBURST	-1665	User already in a send-burst.
QUE_ER_INTERRUPT	-1100	Operation was interrupted.
QUE_ER_ISFROZEN	-1007	A *BlockOpt* of QUE_WAIT or QUE_TIMEOUT() was specified after the instance was frozen by the calling user.
QUE_ER_MSGHDRNOTREMOVED	-1626	*MsgHdr* references a message header that has not been dequeued.
QUE_ER_MSGHDRREMOVED	-1624	*MsgHdr* has been removed from queue.
QUE_ER_NOASYNC	-1006	An asynchronous operation was attempted with no asynchronous environment present.
QUE_ER_NOMORE	-1038	No more queues.
QUE_ER_NOSECCFG	-1655	No [QUESYS] section in ".cfg" file.
QUE_ER_NOSECIDS	-1656	No [QUESYS] section in ".ids" file.
QUE_ER_NOSUBSYSTEM	-1004	QueSys is not configured in the instance.
QUE_ER_NOTEMPTY	-1622	The queue is not empty.
QUE_ER_NOTFOUND	-1033	Queue with *Name* does not exist.
QUE_ER_NOTFROZEN	-1008	QueSys not frozen.
QUE_ER_NOTINSENDBURST	-1663	User not in send-burst.
QUE_ER_NOTLOCAL	-1037	Instance is not local.
QUE_ER_NOTLOGGEDIN	-1002	User not logged into instance (User never logged in, was aborted or disconnected).
QUE_ER_NOWAIT	-1034	*BlockOpt* of QUE_NOWAIT specified and request was not immediately satisfied.
QUE_ER_PURGED	-1621	Another user purged a queue that the blocked QueSend() call was waiting on. The blocked QueSend() operation was cancelled. No message was sent.
QUE_ER_SYSERR	-1101	Send-burst not started due to system error.
QUE_ER_TEXTFULL	-1627	Text space is not available when QUE_REPLICATE or QUE_REPLACE_*XX* is specified, causing

Symbolic Error Name	Number	Description
		call to fail.
QUE_ER_TIMEOUT	-1099	The blocked operation timed out.
QUE_ER_TOOBIG	-1631	The size of the message exceeds the byte capacity of one of the listed Qids (= **QidPtr*).
QUE_ER_TRIGGERNOTEXIST	-1052	Trigger not previously defined
QUE_ER_WAITEDON	-1623	A user is waiting for a message on *Qid*.

5.4.2 QUESYS ERROR CODES: BY MESSAGE NUMBER

Number	Symbolic Error Name	Description
-1002	QUE_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
-1004	QUE_ER_NOSUBSYSTEM	QueSys is not configured in the instance.
-1006	QUE_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.
-1007	QUE_ER_ISFROZEN	A *BlockOpt* of QUE_WAIT or QUE_TIMEOUT() was specified after the instance was frozen by the calling user.
-1008	QUE_ER_NOTFROZEN	QueSys not frozen.
-1014	QUE_ER_BADLISTOFFSET	Invalid offset value specified.
-1019	QUE_ER_BADBUFFER	*MsgBuf* is NULL.
-1022	QUE_ER_BADFILENAME	Invalid *SpoolFileName* specified.
-1023	QUE_ER_BADUID	No user with specified *Uid*.
-1024	QUE_ER_BADVAL	Illegal trigger parameter value.
-1030	QUE_ER_BADOPTION	Invalid *Options* parameter.
-1031	QUE_ER_BADBLOCKOPT	Invalid *BlockOpt*.
-1032	QUE_ER_DUPLICATE	Queue with *Name* already exists.
-1033	QUE_ER_NOTFOUND	Queue with *Name* does not exist.
-1034	QUE_ER_NOWAIT	*BlockOpt* of QUE_NOWAIT specified and request was not immediately satisfied.
-1035	QUE_ER_DESTROYED	[A] Another user destroyed a queue that a blocked QueBurstSend() call was waiting to enqueue onto. The blocked QueBurstSend() operation was canceled. No message was enqueued.
-1037	QUE_ER_NOTLOCAL	Instance is not local.
-1038	QUE_ER_NOMORE	No more queues.
-1051	QUE_ER_BADTRIGGERCODE	Bad trigger code.
-1052	QUE_ER_TRIGGERNOTEXIST	Trigger not previously defined
-1097	QUE_ER_ASYNC	Operation is being performed asynchronously.

Number	Symbolic Error Name	Description
-1098	QUE_ER_ASYNCABORT	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
-1099	QUE_ER_TIMEOUT	The blocked operation timed out.
-1100	QUE_ER_INTERRUPT	Operation was interrupted.
-1101	QUE_ER_SYSERR	Send-burst not started due to system error.
-1610	QUE_ER_BADSID	*Sid* is not a valid semaphore ID.
-1611	QUE_ER_BADQUENAME	Invalid *Name* parameter.
-1612	QUE_ER_BADQID	Bad *TargetQid*, or QUE_NULL_QID was specified when valid Qid is required.
-1613	QUE_ER_BADQIDLIST	Invalid QidList parameter.
-1614	QUE_ER_BADLENGTH	Invalid *MsgLength* parameter.
-1615	QUE_ER_BADLIMIT	Invalid *LimitMsgs* or *LimitBytes* parameter.
-1616	QUE_ER_BADPRIORITY	Invalid *Priority* parameter.
-1617	QUE_ER_BADTEXT	*MsgHdr* has invalid text pointer.
-1618	QUE_ER_BADMSGSELECTCODE	Invalid MsgSelectCode within *QidList*.
-1619	QUE_ER_BADQUESELECTCODE	Invalid QueSelectCode parameter.
-1620	QUE_ER_BADDIRECTION	Invalid *Direction* parameter.
-1621	QUE_ER_PURGED	Another user purged a queue that the blocked QueSend() call was waiting on. The blocked QueSend() operation was cancelled. No message was sent.
-1622	QUE_ER_NOTEMPTY	The queue is not empty.
-1623	QUE_ER_WAITEDON	A user is waiting for a message on *Qid*.
-1624	QUE_ER_MSGHDRREMOVED	*MsgHdr* has been removed from queue.
-1625	QUE_ER_ENDOFQUEUE	An end of the queue has been reached.
-1626	QUE_ER_MSGHDRNOTREMOVED	*MsgHdr* references a message header that has not been dequeued.
-1627	QUE_ER_TEXTFULL	Text space is not available when QUE_REPLICATE or QUE_REPLACE_*XX* is specified, causing call to fail.
-1631	QUE_ER_TOOBIG	The size of the message exceeds the byte capacity of one of the listed Qids (= **QidPtr*).
-1641	QUE_ER_CAPACITY_USER	QueSys user table full.
-1642	QUE_ER_CAPACITY_HEADER	QueSys header table full.
-1643	QUE_ER_CAPACITY_NODE	QueSys node table full.
-1644	QUE_ER_CAPACITY_TABLE	Queue table full.

Number	Symbolic Error Name	Description
-1645	QUE_ER_CAPACITY_ASYNC_USER	QueSys async user table full.
-1651	QUE_ER_FAILSTART	QueSys initialization failed.
-1652	QUE_ER_FAILSTOP	QueSys termination failed.
-1653	QUE_ER_GHOSTSTART	Cannot register QueSys with *X♦IPC* object daemon.
-1654	QUE_ER_GHOSTSTOP	Cannot deregister QueSyswith *X♦IPC* object daemon.
-1655	QUE_ER_NOSECCFG	No [QUESYS] section in ".cfg" file.
-1656	QUE_ER_NOSECIDS	No [QUESYS] section in ".ids" file.
-1663	QUE_ER_NOTINSENDBURST	User not in send-burst.
-1665	QUE_ER_INSENDBURST	User already in a send-burst.
-1666	QUE_ER_INRECEIVEBURST	User is in a receive-burst.
-1667	QUE_ER_BADERROROPT	Invalid *ErrorOption* parameter.
-1671	QUE_ER_BADREADAHEAD	Invalid ReadAheadBufSize parameter.
-1674	QUE_ER_BADSYNCMODE	Invalid *Mode* parameter.

5.4.3 SEMSYS ERROR CODES: BY SYMBOLIC ERROR NAME

SYMBOLIC ERROR NAME	NUMBER	DESCRIPTION
SEM_ER_ASYNC	-1097	Operation is being performed asynchronously.
SEM_ER_ASYNCABORT	-1098	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
SEM_ER_BADACQUIRETYPE	-1317	Invalid *AcquireType* parameter.
SEM_ER_BADBLOCKOPT	-1031	Invalid *BlockOpt*.
SEM_ER_BADCLEAROPT	-1315	Invalid *ClearOpt*.
SEM_ER_BADLISTOFFSET	-1014	Invalid offset value specified.
SEM_ER_BADOPTION	-1030	Invalid *Option* parameter.
SEM_ER_BADSEMNAME	-1311	Invalid *Name* parameter.
SEM_ER_BADSEMVALUE	-1312	Invalid *CreateValue* parameter.
SEM_ER_BADSID	-1314	*SidList* contains a bad Sid. **RetSid* is set to the invalid Sid.
SEM_ER_BADSIDLIST	-1313	Bad *SidList*.
SEM_ER_BADUID	-1023	Invalid *Uid* parameter.
SEM_ER_BADWAITTYPE	-1316	Invalid *WaitType* parameter.
SEM_ER_CANCEL	-1331	Another user issued a SemCancel() call for one of the semaphores in *SidList*. The blocked SemWait() operation was cancelled. **RetSid* is set to the Sid of the semaphore for which the SemCancel() was issued.
SEM_ER_CAPACITY_ASYNC_USER	-1344	SemSys async user table full.
SEM_ER_CAPACITY_NODE	-1342	SemSys node table full.
SEM_ER_CAPACITY_NODE	-1342	SemSys node table full.
SEM_ER_CAPACITY_TABLE	-1343	Semaphore table full.
SEM_ER_CAPACITY_USER	-1341	SemSys user table full.
SEM_ER_DESTROYED	-1035	Another user destroyed a semaphore that was being waited on by this user. The blocked acquire operation was cancelled. **RetSid* is set to the Sid of the destroyed semaphore.
SEM_ER_DUPLICATE	-1032	Semaphore with *Name* already exists.
SEM_ER_FAILSTART	-1351	SemSys initialization failed.
SEM_ER_FAILSTOP	-1352	SemSys termination failed.
SEM_ER_GHOSTSTART	-1353	Cannot register SemSys with X♦IPC object daemon.
SEM_ER_GHOSTSTOP	-1354	Cannot deregister SemSys with X♦IPC object daemon.

SYMBOLIC ERROR NAME	NUMBER	DESCRIPTION
SEM_ER_INTERRUPT	-1100	The blocked operation was interrupted by an asynchronous event (such as a signal). The operation has been canceled.
SEM_ER_INVALIDSID	-1318	Invalid semaphore identifier specified.
SEM_ER_ISFROZEN	-1007	A *BlockOpt* of SEM_WAIT or SEM_TIMEOUT() was specified after the instance was frozen by the calling user.
SEM_ER_NOASYNC	-1006	An asynchronous operation was attempted with no asynchronous environment present.
SEM_ER_NOMORE	-1038	No more data.
SEM_ER_NOSECCFG	-1355	No [SEMSYS] section in ".cfg" file.
SEM_ER_NOSECIDS	-1356	No [SEMSYS] section in ".ids" file.
SEM_ER_NOSUBSYSTEM	-1004	SemSys is not configured in the instance.
SEM_ER_NOTFOUND	-1033	Semaphore with *Name* does not exist.
SEM_ER_NOTFROZEN	-1003	SemSys not frozen.
SEM_ER_NOTLOGGEDIN	-1002	User not logged into instance (User never logged in, was aborted or disconnected).
SEM_ER_NOWAIT	-1034	*BlockOpt* of SEM_NOWAIT was specified and the request was not immediately satisfied.
SEM_ER_SEMBUSY	-1321	Semaphore *Sid* held or blocked on by other users.
SEM_ER_SEMCLEAR	-1324	*SidList* contains a Sid of a semaphore which is already clear. **RetSid* is set to that Sid.
SEM_ER_SEMNOTHELD	-1322	*SidList* contains a Sid of a semaphore not currently held by the user. **RetSid* is set to that Sid.
SEM_ER_SEMSET	-1323	*SidList* contains a Sid of a semaphore which is already set. **RetSid* is set to that Sid.
SEM_ER_SYSERR	-1101	An internal error has occurred while processing the request.
SEM_ER_TIMEOUT	-1099	The time out period for the blocked operation has expired without satisfying the request.

5.4.4 SEMSYS ERROR CODES: BY MESSAGE NUMBER

NUMBER	SYMBOLIC ERROR NAME	DESCRIPTION
-1002	SEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
-1003	SEM_ER_NOTFROZEN	SemSys not frozen.
-1004	SEM_ER_NOSUBSYSTEM	SemSys is not configured in the instance.
-1006	SEM_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.
-1007	SEM_ER_ISFROZEN	A *BlockOpt* of SEM_WAIT or SEM_TIMEOUT() was specified after the instance was frozen by the calling user.
-1014	SEM_ER_BADLISTOFFSET	Invalid offset value specified.
-1023	SEM_ER_BADUID	Invalid *Uid* parameter.
-1030	SEM_ER_BADOPTION	Invalid *Option* parameter.
-1031	SEM_ER_BADBLOCKOPT	Invalid *BlockOpt*.
-1032	SEM_ER_DUPLICATE	Semaphore with *Name* already exists.
-1033	SEM_ER_NOTFOUND	Semaphore with *Name* does not exist.
-1034	SEM_ER_NOWAIT	*BlockOpt* of SEM_NOWAIT was specified and the request was not immediately satisfied.
-1035	SEM_ER_DESTROYED	Another user destroyed a semaphore that was being waited on by this user. The blocked acquire operation was cancelled. **RetSid* is set to the Sid of the destroyed semaphore.
-1038	SEM_ER_NOMORE	No more data.
-1097	SEM_ER_ASYNC	Operation is being performed asynchronously.
-1098	SEM_ER_ASYNCABORT	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
-1099	SEM_ER_TIMEOUT	The time out period for the blocked operation has expired without satisfying the request.
-1100	SEM_ER_INTERRUPT	The blocked operation was interrupted by an asynchronous event (such as a signal). The operation has been canceled.
-1101	SEM_ER_SYSERR	An internal error has occurred while processing the request.
-1311	SEM_ER_BADSEMNAME	Invalid *Name* parameter.
-1312	SEM_ER_BADSEMVALUE	Invalid *CreateValue* parameter.
-1313	SEM_ER_BADSIDLIST	Bad *SidList*.
-1314	SEM_ER_BADSID	*SidList* contains a bad Sid. **RetSid* is set to the invalid Sid.
-1315	SEM_ER_BADCLEAROPT	Invalid *ClearOpt*.
-1316	SEM_ER_BADWAITTYPE	Invalid *WaitType* parameter.

NUMBER	SYMBOLIC ERROR NAME	DESCRIPTION
-1317	SEM_ER_BADACQUIRETYPE	Invalid *AcquireType* parameter.
-1318	SEM_ER_INVALIDSID	Invalid semaphore identifier specified.
-1321	SEM_ER_SEMBUSY	Semaphore *Sid* held or blocked on by other users.
-1322	SEM_ER_SEMNOTHELD	*SidList* contains a Sid of a semaphore not currently held by the user. **RetSid* is set to that Sid.
-1323	SEM_ER_SEMSET	*SidList* contains a Sid of a semaphore which is already set. **RetSid* is set to that Sid.
-1324	SEM_ER_SEMCLEAR	*SidList* contains a Sid of a semaphore which is already clear. **RetSid* is set to that Sid.
-1331	SEM_ER_CANCEL	Another user issued a SemCancel() call for one of the semaphores in *SidList*. The blocked SemWait() operation was cancelled. **RetSid* is set to the Sid of the semaphore for which the SemCancel() was issued.
-1341	SEM_ER_CAPACITY_USER	SemSys user table full.
-1342	SEM_ER_CAPACITY_NODE	SemSys node table full.
-1343	SEM_ER_CAPACITY_TABLE	Semaphore table full.
-1344	SEM_ER_CAPACITY_ASYNC_USER	SemSys async user table full.
-1351	SEM_ER_FAILSTART	SemSys initialization failed.
-1352	SEM_ER_FAILSTOP	SemSys termination failed.
-1353	SEM_ER_GHOSTSTART	Cannot register SemSys with *X♦IPC* object daemon.
-1354	SEM_ER_GHOSTSTOP	Cannot deregister SemSys with *X♦IPC* object daemon.
-1355	SEM_ER_NOSECCFG	No [SEMSYS] section in ".cfg" file.
-1356	SEM_ER_NOSECIDS	No [SEMSYS] section in ".ids" file.

5.4.5 MEMSYS ERROR CODES: BY SYMBOLIC ERROR NAME

SYMBOLIC ERROR NAME	NUMBER	DESCRIPTION
MEM_ER_ACCESSDENIED	-1936	Specified *Section* is currently owned by another user.
MEM_ER_ALREADYLOCKED	-1935	*MidList* contains a memory section that is already locked by the user. **SecPtr* identifies the invalid sectionl
MEM_ER_ASYNC	-1097	Operation is being performed asynchronously.
MEM_ER_ASYNCABORT	-1098	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
MEM_ER_BADBLOCKOPT	-1031	Invalid *BlockOpt*.
MEM_ER_BADBUFFER	-1019	*Buffer* is NULL.
MEM_ER_BADLISTOFFSET	-1014	Invalid offset value specified.
MEM_ER_BADLOCKTYPE	-1010	Invalid *LockType* parameter.
MEM_ER_BADMID	-1912	Invalid Memory Segment ID *Mid*.
MEM_ER_BADMIDLIST	-1913	Invalid MidList parameter.
MEM_ER_BADOPTION	-1030	Invalid *Options* parameter.
MEM_ER_BADOWNTYPE	-1918	Invalid *OwnType* parameter.
MEM_ER_BADPRIVILEGE	-1920	Invalid *OwnerPrivilege* or *OtherPrivilege* parameter(s).
MEM_ER_BADSECTION	-1916	*MidList* contains a bad section. **RetSec* identifies the invalid section.
MEM_ER_BADSEGNAME	-1911	Invalid *Name* parameter.
MEM_ER_BADSID	-1910	*Sid* is not a valid semaphore ID.
MEM_ER_BADSIZE	-1915	Invalid *Size* parameter.
MEM_ER_BADTARGET	-1914	Invalid target specification.
MEM_ER_BADTRIGGERCODE	-1051	Bad trigger code
MEM_ER_BADUID	-1023	Invalid *AUid* parameter.
MEM_ER_BADVAL	-1024	Illegal trigger parameter value
MEM_ER_CAPACITY_ASYNC_USER	-1946	MemSys async user table full.
MEM_ER_CAPACITY_NODE	-1943	MemSys node table full.
MEM_ER_CAPACITY_NODE	-1943	MemSys node table full.
MEM_ER_CAPACITY_POOL	-1942	MemSys text pool full.
MEM_ER_CAPACITY_SECTION	-1944	MemSys section table full.
MEM_ER_CAPACITY_TABLE	-1945	MemSys segment table full.

SYMBOLIC ERROR NAME	NUMBER	DESCRIPTION
MEM_ER_DESTROYED	-1035	Another user destroyed the memory segment targeted by the blocked MemWrite operation.
MEM_ER_DUPLICATE	-1032	Section already exists.
MEM_ER_FAILSTART	-1951	MemSys initialization failed.
MEM_ER_FAILSTOP	-1952	MemSys termination failed.
MEM_ER_GHOSTSTART	-1953	Cannot register MemSys with *X♦IPC* object daemon.
MEM_ER_GHOSTSTOP	-1954	Cannot deregister MemSys with *X♦IPC* object daemon.
MEM_ER_INTERRUPT	-1100	Operation was interrupted.
MEM_ER_ISFROZEN	-1007	A *BlockOpt* of MEM_WAIT or MEM_TIMEOUT() was specified after the instance was frozen by the calling user.
MEM_ER_MEMBUSY	-1933	MemSys Segment has one or more sections defined over it.
MEM_ER_NOASYNC	-1066	An asynchronous operation was attempted with no asynchronous environment present.
MEM_ER_NOMORE	-1038	No more data.
MEM_ER_NOSECCFG	-1955	No [MEMSYS] section in ".cfg" file.
MEM_ER_NOSECIDS	-1956	No [MEMSYS] section in ".ids" file.
MEM_ER_NOSUBSYSTEM	-1004	MemSys is not configured in the instance.
MEM_ER_NOTFOUND	-1033	Memory Segment with *Name* does not exist.
MEM_ER_NOTFROZEN	-1008	MemSys not frozen.
MEM_ER_NOTLOCAL	-1037	Instance is not local.
MEM_ER_NOTLOCKED	-1932	*MidList* contains a memory section not currently locked by the user. **RetSec* identifies the invalid section.
MEM_ER_NOTLOGGEDIN	-1002	User not logged into instance (User never logged in, was aborted or disconnected).
MEM_ER_NOTOWNER	-1931	*MidList* contains a memory section not currently owned by the user. **RetSec* identifies the invalid section.
MEM_ER_NOWAIT	-1934	*BlockOpt* of MEM_NOWAIT was specified and the request was not immediately satisfied.
MEM_ER_SYSERR	-1101	An internal error has occurred while processing the request.
MEM_ER_TIMEOUT	-1099	The time out period for the blocked lock operation has expired without satisfying the request.
MEM_ER_TRIGGERNOTEXIST	-1052	Trigger not previously defined

5.4.6 MEMSYS ERROR CODES: BY MESSAGE NUMBER

NUMBER	ERROR CODE	DESCRIPTION
-1002	MEM_ER_NOTLOGGEDIN	User not logged into instance (User never logged in, was aborted or disconnected).
-1004	MEM_ER_NOSUBSYSTEM	MemSys is not configured in the instance.
-1007	MEM_ER_ISFROZEN	A *BlockOpt* of MEM_WAIT or MEM_TIMEOUT() was specified after the instance was frozen by the calling user.
-1008	MEM_ER_NOTFROZEN	MemSys not frozen.
-1010	MEM_ER_BADLOCKTYPE	Invalid *LockType* parameter.
-1014	MEM_ER_BADLISTOFFSET	Invalid offset value specified.
-1019	MEM_ER_BADBUFFER	*Buffer* is NULL.
-1023	MEM_ER_BADUID	Invalid *AUid* parameter.
-1024	MEM_ER_BADVAL	Illegal trigger parameter value
-1030	MEM_ER_BADOPTION	Invalid *Options* parameter.
-1031	MEM_ER_BADBLOCKOPT	Invalid *BlockOpt*.
-1032	MEM_ER_DUPLICATE	Section already exists.
-1033	MEM_ER_NOTFOUND	Memory Segment with *Name* does not exist.
-1035	MEM_ER_DESTROYED	Another user destroyed the memory segment targeted by the blocked MemWrite operation.
-1037	MEM_ER_NOTLOCAL	Instance is not local.
-1038	MEM_ER_NOMORE	No more data.
-1051	MEM_ER_BADTRIGGERCODE	Bad trigger code
-1052	MEM_ER_TRIGGERNOTEXIST	Trigger not previously defined
-1066	MEM_ER_NOASYNC	An asynchronous operation was attempted with no asynchronous environment present.
-1097	MEM_ER_ASYNC	Operation is being performed asynchronously.
-1098	MEM_ER_ASYNCABORT	Asynchronous operation aborted before completion. This error code is not returned by the function call. It is set in the Asynchronous Result Control Block *RetCode* field.
-1099	MEM_ER_TIMEOUT	The time out period for the blocked lock operation has expired without satisfying the request.
-1100	MEM_ER_INTERRUPT	Operation was interrupted.
-1101	MEM_ER_SYSERR	An internal error has occurred while processing the request.
-1910	MEM_ER_BADSID	*Sid* is not a valid semaphore ID.
-1911	MEM_ER_BADSEGNAME	Invalid *Name* parameter.
-1912	MEM_ER_BADMID	Invalid Memory Segment ID *Mid*.
-1913	MEM_ER_BADMIDLIST	Invalid MidList parameter.

NUMBER	ERROR CODE	DESCRIPTION
-1914	MEM_ER_BADTARGET	Invalid target specification.
-1915	MEM_ER_BADSIZE	Invalid *Size* parameter.
-1916	MEM_ER_BADSECTION	*MidList* contains a bad section. **RetSec* identifies the invalid section.
-1918	MEM_ER_BADOWNTYPE	Invalid *OwnType* parameter.
-1920	MEM_ER_BADPRIVILEGE	Invalid *OwnerPrivilege* or *OtherPrivilege* parameter(s).
-1931	MEM_ER_NOTOWNER	*MidList* contains a memory section not currently owned by the user. **RetSec* identifies the invalid section.
-1932	MEM_ER_NOTLOCKED	*MidList* contains a memory section not currently locked by the user. **RetSec* identifies the invalid section.
-1933	MEM_ER_MEMBUSY	MemSys Segment has one or more sections defined over it.
-1934	MEM_ER_NOWAIT	*BlockOpt* of MEM_NOWAIT was specified and the request was not immediately satisfied.
-1935	MEM_ER_ALREADYLOCKED	*MidList* contains a memory section that is already locked by the user. **SecPtr* identifies the invalid sectionl
-1936	MEM_ER_ACCESSDENIED	Specified *Section* is currently owned by another user.
-1942	MEM_ER_CAPACITY_POOL	MemSys text pool full.
-1943	MEM_ER_CAPACITY_NODE	MemSys node table full.
-1944	MEM_ER_CAPACITY_SECTION	MemSys section table full.
-1945	MEM_ER_CAPACITY_TABLE	MemSys segment table full.
-1946	MEM_ER_CAPACITY_ASYNC_USER	MemSys async user table full.
-1951	MEM_ER_FAILSTART	MemSys initialization failed.
-1952	MEM_ER_FAILSTOP	MemSys termination failed.
-1953	MEM_ER_GHOSTSTART	Cannot register MemSys with *X•IPC* object daemon.
-1954	MEM_ER_GHOSTSTOP	Cannot deregister MemSys with *X•IPC* object daemon.
-1955	MEM_ER_NOSECCFG	No [MEMSYS] section in ".cfg" file.
-1956	MEM_ER_NOSECIDS	No [MEMSYS] section in ".ids" file.