Envoy XIPC

Version 3.4.0

QUESYS/SEMSYS/MEMSYS

USER GUIDE

Envoy Technologies Inc.

555 Route 1 South
Iselin, NJ 08830

http://www.envoytech.com

Printed in United States of America

X•IPC VERSION 3.4.0

QUESYS/MEMSYS/SEMSYS

USER GUIDE

Table of Contents

1. INTRODUCTION

1.1 Purpose

This document presents User and Programming guidance for Version 3.0 of the QueSys, MemSys and SemSys subsystems of *X♦IPC,* the Extended Interprocess Communication Facilities product from Momentum Software Corporation.
X♦IPC is a tool kit for developing software systems employing Interprocess Communication (IPC). *X♦IPC* comprises the following subsystems:

- QueSys, the Message Queue System
- MemSys, the Shared Memory System
- SemSys, the Semaphore System
- MomSys, the Message Oriented Middleware System

The present document describes the concepts, configuration and programming considerations relevant to the first three subsystems; there is a companion Reference Guide for these three subsystems. MomSys is documented in its own User Guide and Reference Manual.
In addition, there is separate documentation for the over-all X♦IPC product, including an X♦IPC User Guide and an X♦IPC Reference Manual; these two system-level documents can be considered prerequisite to the subsystem documentation and should be read first, in order to become familiar with general X♦IPC concepts and for general programming guidance.
X♦IPC is a set of libraries and support utilities that greatly simplifies software development efforts involving stand-alone and network IPC. Used together or individually, *X♦IPC* subsystems provide significant enhancements to the native IPC facilities of the supported operating systems. *X♦IPC* provides the systems developer with a state-of-the-art IPC development environment, including: on-line interactive IPC monitoring and debugging; extended basic and advanced functionality; immediate inter-operating system IPC source-code portability; guaranteed message delivery; complete network transparency; and dynamic configuration.

1.1.1 X♦IPC Subsystems

X♦IPC is comprised of four IPC subsystems, each of which includes a library of functions and support utilities; the first three subsystems are addressed in the present document:

- QueSys, the Message Queue System

The *X♦IPC* message queue system is known as QueSys. QueSys is a complete memory-based, high-performance message queuing facility. Many advanced features are included (e.g., individualized queue sizing, dynamic queue spooling, queue multiplexing, etc.) to facilitate most necessary message queuing requirements.

- MemSys, the Shared Memory System

The *X♦IPC* shared memory system is known as MemSys. MemSys is a complete shared-memory management system. It includes memory allocation as well as access control, synchronization, locking and protection at the byte level.

- SemSys, the Semaphore System

The X♦IPC semaphore subsystem is known as SemSys. SemSys includes a comprehensive implementation of event and resource semaphores. Its wide range of operations and the various waiting and acquiring alternatives ensures that almost every semaphore-related system requirement can be easily implemented.

- MomSys, the Message Oriented Middleware subsystem

The X♦IPC message oriented middleware subsystem is known as MomSys. MomSys is a highly scalable, dynamically configurable, guaranteed message delivery facility. Ideal for mission-critical, enterprise-wide applications, MomSys ensures the constant trackability of all messages.

1.2 Scope

This QueSys/MemSys/SemSys User Guide is for experienced software developers, who are familiar with the basic concepts of IPC as well as with common software development practices, and who need the enhancements provided by X♦IPC for easing and expediting their development of quality, portable, multi-tasking or distributed applications.

For ease of use and to facilitate presentation, this volume repeats some material that appears as well in the X♦IPC User Guide; as noted above, the X♦IPC User Guide should be read before the present document.

1.3 Documentation Road Map

The following publications are available to support X♦IPC Version 3.4.0:

- Getting Started with X♦IPC is a brief introduction to the product which gives the user a "fast track" to select the relevant documentation, install the software and rapidly begin using X♦IPC .
- X♦IPC Platform Notes provide platform-specific information regarding product installation, program compilation, program linking and, where appropriate, configuration and administration guidance. The supported environments are individually documented

X♦IPC system level documentation:

- The X♦IPC User Guide describes in detail how to employ X♦IPC for distributed application development. This document is generic in that it presents X♦IPC without regard to any particular hardware platform, operating system or network protocol. The information is presented at an X♦IPC -system-level, i.e., it is X♦IPC -subsystem-independent.
- The X♦IPC Reference Manual provides X♦IPC (system level) commands, functions and macros, as well as function calling sequences and possible return codes. Included are code segments and sample programs.

QueSys/MemSys/SemSys documentation:

- The QueSys/MemSys/SemSys User Guide–*this document*–describes in detail how to use these three X♦IPC subsystems for distributed application development. It includes API descriptions as well as topical presentations on special subsystem features.

- The QueSys/MemSys/SemSys Reference Manual details subsystem-level parameters, functions and macros, interactive commands and sample programs, as well function calling sequences and possible return codes.

MomSys documentation

- The MomSys User Guide describes in detail how to use the MomSys subsystem for distributed application development. It includes API descriptions as well as topical presentations on special subsystem features.
- The MomSys Reference Manual details subsystem-level parameters, functions and macros, interactive commands and sample programs, as well function calling sequences and possible return codes.

2. QUESYS: THE *X♦IPC* MESSAGE QUEUE SYSTEM

2.1 QueSys Concepts

To understand *X♦IPC* QueSys, the user should be familiar with some basic QueSys concepts. The following sections introduce these ideas.

2.1.1 Message Queues

The most important abstraction within QueSys is that of a QueSys message queue. QueSys queues are memory-based, high-performance mechanisms for supporting network transparent message-based distributed applications.

2.1.1.1 Time and Priority Ordering

A QueSys message queue is a set of messages that are ordered both by "arrival time" and "message priority." The two orderings are referred to as the time and priority "strands" of the queue.

In a sense, a message queue has two personalities. Traversing the queue chronologically one encounters its messages in the order in which they were placed on the queue. Traversing the queue by priority presents the messages in order of decreasing priority. Of course, each message actually exists only once on the queue.

2.1.1.2 Message Headers

Beyond the issue of queue strands, there is the question of what exactly is a QueSys message? In fact, the objects actually on a queue are not messages at all but rather data structures called "message headers."

A QueSys message is composed of two components:

- A message header structure that moves about on QueSys queues
- The actual message text that resides in a message text pool area

A message header is a structure that completely describes a QueSys message. It contains all relevant information about a message, including:

- The message's user assigned priority
- The message's time of arrival on the queue

- The length of the message text
- A pointer to the message text

Breaking up messages into separate "header" and "text" components is efficient for a number of reasons. For one, the management of a message's text space is independent of which queue the message is currently on.

More important, messages can be manipulated and moved among queues without any need to copy the message's text as part of each move. The only portion that needs to be moved from queue to queue is the message header. The message text can remain in one place throughout the message's existence.

Message 'A' can be transferred from queue 0 to queue 1 by moving the header alone and not touching its text.

X•IPC lets the developer manipulate messages at two levels:

- At a lower level: *headers and texts separately*.

The programmer views message headers and their texts as separate items under his control. A message is dispatched to a queue in two steps using this approach:

- First, the text is written into the QueSys message text pool. This produces a message header that references the written text.
- Then, the message header is placed onto the appropriate message queue.

For message retrieval, the process is reversed:

- First, the appropriate message header is removed from a queue.
- Then, its corresponding text is read.

In fact, a message header can be moved on and off numerous queues before its text is actually read (i.e., purged) from the text pool. This can be an important consideration for an application that employs complex message routing schemes, before messages actually get "consumed." This is especially important if large-sized messages are involved.

☐ At a higher level: *headers and texts as single units*.

Message dispatch and retrieval operations are accomplished as single operations with the details of message headers and text pool reads and writes hidden from the programmer.

The message dispatch operation writes the user's text to the message text pool and then immediately places the header onto the appropriate queue.

The message retrieval operation accesses the desired header and then immediately reads its text out of the text pool.

The choice of approach depends on the level of flexibility needed at a particular point in a program. It will sometimes make sense to mix the two methods, using, for instance, high level operations at the terminals of a message's journey and low level header operations for intermediate inter-queue moves.

2.1.1.3 Message Text

A message header contains two fields that relate to a message's text: a message length field and a pointer to the actual text. These two fields provide the necessary information for working with a message's text.

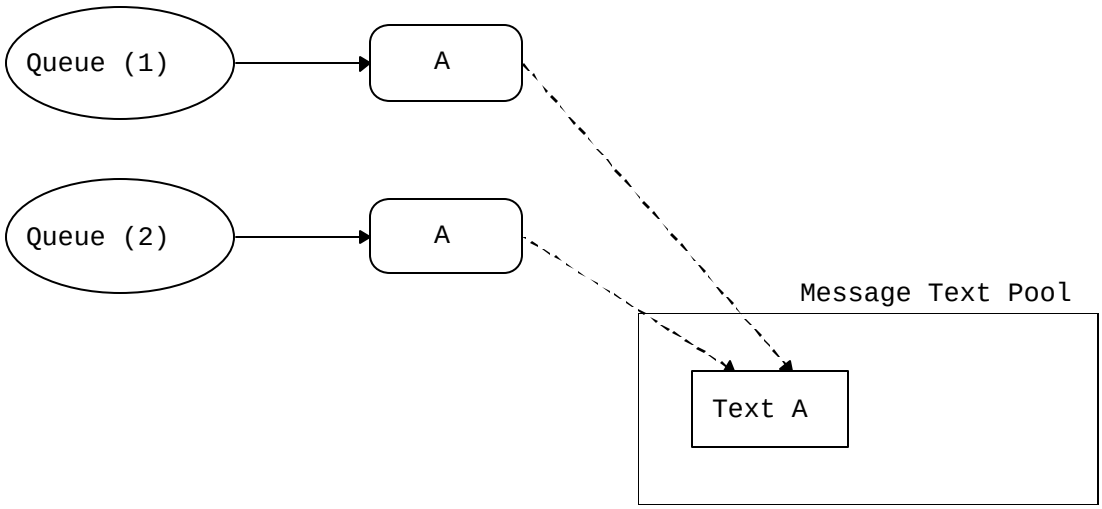
Obviously, using the text pointer to modify a message's text while it is still resident in the message text pool is fraught with danger. If the message's header is still on a queue, then there is no guarantee that it will remain there. And, if the header has been removed from a queue, “invading” the instance's message text pool is an indication of poor design, at best.

An even greater danger arises when working within a network instance. In that case, there is no certainty that the text pointer contained in the message header references a text pool on the local machine. It may in fact refer to a text pool residing on a different network node.

For these reasons, it is recommended that a program generally avoid attempting to directly modify a message's text once it has been written to the message text pool. However, examining all or part of a message's text via its message header can be accomplished using the QuePointer() and QueCopy() functions. This can be very useful for determining the importance of a message to a program without removing the entire text from the text pool.

2.1.1.4 Multiple Text References

One of the more interesting features of *X•IPC* 's QueSys is that it allows a single message-text block (i.e., a single allocated text-pool block) to be referenced by more than one QueSys message header.
This can be useful when there is a need to move a large message through two separate queues (e.g., replication to separate server programs). Using QueSys, it is possible to set up a single text-pool block with the data and to have multiple headers reference the block, as in the following diagram:



QueSys automatically manages the reference counts to the text-pool blocks and frees it when the last reference is removed.
An example of this mechanism's use is when QueSend() or QuePut() are employed with the `QUE_REPLICATE` option for multicasting message copies to multiple recipients via a single QueSys function call. QueSys verbs that can cause such multiple referencing to occur are: QueSend(), QuePut() and QueMsgHdrDup(). Refer to each of the respective descriptions in this Guide and in the accompanying Reference Manual.

2.1.1.5 Message Priority

A priority is assigned to a message when the message is placed onto a queue. The priority value is specified by the user as part of the message dispatch operation.
As stated earlier, a queue's messages are kept in priority order via the queue's priority strand. Every QueSys message has a priority value associated with it. QueSys priorities are long positive integers.
Priorities are most useful for selective message retrieval operations. A wide range of priority specification capabilities is available for retrieving a message from a queue.
These include:

- Specific priority (e.g., priority == 100).
- Priority range specification (e.g., priority between 200 and 300).
- Basic boolean operations (e.g., priority != 25, priority < 100).
- Extreme values (e.g., maximum or minimum priority on a queue).

In addition, a number of QueView monitor capabilities refer to message priority. For example, browsing a queue's messages can proceed in priority order; watching front and rear messages of a queue can be done from a priority perspective, etc. These capabilities are described later in this chapter.

2.1.1.6 Message Time/Sequence Stamp

Messages on a queue are also kept in chronological order via the queue's time strand. Each message is automatically stamped with an internal sequence number when it is placed onto a queue. This relative "time stamp" value is used for keeping messages on queues ordered by arrival. Each message queue has its own sequencing of messages. The sequence number of the first message on a queue is "one." Every subsequent message on that queue receives the next sequential number.

Sequence numbers can be used as a key for selecting messages from a queue. We will see later that QueSys provides the means for selecting a particular message from a queue, based on its sequence number. This is discussed in detail in the Advanced Topics chapter.

Here, too, QueView refers to message chronology in a number of ways. Browsing a queue's messages can proceed chronologically; viewing front and rear messages of a queue can be done from a message arrival perspective, etc.

One QueSys operation that makes particular use of a message's "time stamp" is the QueUnget() function. QueUnget() returns a message to a queue, placing it chronologically precisely where it was taken from. This is accomplished using the message's "time stamp."

2.1.2 Message Queue Capacity

An important aspect of *X•IPC* QueSys is its approach to queue sizing. A number of innovations are introduced in this area.

2.1.2.1 Individual Queue Capacity Specifications

QueSys queues are created with individualized capacity limits. The capacity of each queue is specified as part of the queue create operation.

A queue's capacity can be specified in one of four way:

- Maximum messages, maximum bytes.
- Maximum messages, unlimited bytes.
- Unlimited messages, maximum bytes.
- Unlimited messages, unlimited bytes.

Using this capability, it is possible to throttle message traffic within an application. Runaway message-roducing programs can be controlled by setting a limit on the amount of traffic their outbound queues can hold–in terms of messages, bytes or both. This also allows the programmer to protect an instance's message text pool from being monopolized by the same over-producing programs.

Specific queue capacity limits provide the flexibility and control needed within queue-intensive applications.

2.1.2.2 Dynamic Queue Overflow Spooling

Complementing the specific sizing capability is the ability to start overflow message spooling on a queue-by-queue basis.

When spooling for a queue is "on," the queue can absorb messages beyond its normal capacity limitations. This elasticity is made possible via an overflow message spooling mechanism. Messages attempting to enter the full queue are temporarily spooled out to disk until space on the queue becomes available, at which time the messages are automatically absorbed into the queue.
Using this option, a system can impose a memory usage cap on a message queue (via queue capacity limits) even if its producing programs cannot be blocked from producing messages. A network feed such as a stock ticker is one such example. Heavy bursts of messages are not lost. Instead, they are temporarily kept on their destination queue's overflow spool.
Overflow spooling can be started and stopped dynamically by program control, in response to changing traffic loads.
Details of the spooling mechanism are described later in this chapter.

2.1.3 Message Text Pool

The actual text of every QueSys message spends its entire existence in the instance's message text pool. Configuring an instance's message text pool properly can make the difference between a good system and one that performs poorly.
There are two aspects to message text pool configuration:

- The size of the pool.
- The allocation unit used by the pool.

2.1.3.1 Sizing

The message text pool size defines the total amount of memory allocated to the instance for holding QueSys messages (i.e., their text). It is important that the given value be reasonably close to the actual memory requirement.
Too large a value will result in wasted memory; too small a value will result in poorly performing programs. Programs requiring allocations of pool space that are not available will usually wait until the required memory blocks become free. It may be convenient to depend on such a contingency, but this should not be allowed to occur regularly.
The formulae for calculating an efficient message text pool size are presented later in this chapter.

2.1.3.2 Fragmentation

The second component of message text pool configuration is the size of the allocation unit. This value specifies the multiple by which all text pool allocations are made. It directly impacts the level of fragmentation that occurs within the pool.
An instance working with large messages will benefit from an equally large value for its allocation unit size. Wasteful fragmentation will thus be limited. An instance supporting the manipulation of small sized messages will similarly benefit from a small allocation unit size.
The ability to customize an instance's QueSys according to the specific needs of its client programs is one of the important benefits of using X•IPC.
Formulae for determining a proper allocation unit size are presented later in this chapter.

2.1.4 Message Multicasting

QueSys supports a variety of means for building scalable, high-performance message multicasting applications. These methods are described in the Advanced Topics chapter later in this Guide.
Typical applications built using the QueSys multicasting feature include: "publish and subscribe" and "message replication" applications. Numerous such applications of these kinds have been successfully deployed since this capability was introduced within *X•IPC*.

2.1.5 QueBurst() - Ultra-High Throughput Messaging

QueSys provides a specialized set of functions for building distributed applications that require extremely high message throughput performance. These are called the QueSys QueBurst() functions. They are described later in this Guide's Advanced Topics section, "The Queue-Burst Facility for Very High Throughput Message Queuing."
With proper utilization, QueBurst() message throughput over a network can exceed 70% of the protocol's physical bandwidth. Real-time message transfer applications are ideal candidates for QueBurst().

2.1.6 RPC-like Request-Response Inquiry

QueSys includes a specialized function for optimizing the network performance of request-response classes of applications.
The QueSendReceive() function provides this capability and is described later in this Guide.

2.1.7 Queue Multiplexing

X•IPC QueSys provides the developer with a great deal of flexibility in dispatching and retrieving QueSys messages. Of particular note is the ability to operate on multiple message queues atomically. For example, one can retrieve the highest priority message from a group of three queues. Multiplexing distinguishes *X•IPC* QueSys from most other message queuing facilities.
Operations that move messages to and from queues are generalized to work with lists of queues. Queue lists are used in a way that is very similar to the usage of semaphore lists in SemSys. Here, too, an operation involving only one queue uses a single-element list. Ultimately, QueSys message transfer functions move exactly one message to or from exactly one queue. The determination of which queue from the list is selected as the operative queue, and which message is returned, is based at run-time on user-specified arguments to the various function calls.

There are two classes of specification codes:

- Queue Select Codes.
- Message Select Codes.

2.1.7.1 Queue Select Codes

Queue Select Codes (QSC) are codes that select one queue from a list of queues based on certain criteria. A QSC is provided as an argument to QueSys message transfer operations involving queue lists.

As an example, consider a programmer wishing to send a message onto the shortest queue of the group 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" Queue Select Code (i.e., QUE_Q_SHQ) as an argument to the dispatch operation. This selection can be summarized as:

```
QUE_Q_SHQ {a, b, c}
```

Similarly, sending a message onto the longest queue from the list would use the "Longest Queue" QSC (i.e., QUE_Q_LNQ), and would be summarized as:

```
QUE_Q_LNQ {a, b, c}
```

The list of possible Queue Select Codes is extensive. The QSC values that can be used within message "dispatch" operations 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).

Queue Select Codes have a slightly different function when used within message "retrieval" operations. They work together with the second class of codes, Message Select Codes (MSC), to identify the message to be retrieved.

2.1.7.2 Message Select Codes

Retrieving messages within QueSys can be viewed as occurring in two steps.
First, a list of message queues is defined by the program. As part of this definition, a particular message is designated as the candidate message from each of the listed queues. This designation is done using a Message Select Code (MSC).

For example, the list:

```
{ QUE_M_HP(a), QUE_M_EA(b), QUE_M_LA(c) }
```

defines a list of three queues **a**, **b**, and **c** with Message Select Codes that designate:

- The highest priority message on queue **a**: QUE_M_HP(a).
- The earliest arrived message on queue **b**: QUE_M_EA(b).
- The latest arrived message on queue **c**: QUE_M_LA(c)

as their respective candidate messages.
One of the competing candidate messages is then selected from the group based on a Queue Select Code. The chosen message is retrieved and returned to the user.

The *QueSelectCode*s that are valid within message retrieval functions (QueGet() and QueReceive()) can be based on Message Attributes or Queue Attributes.

Based on Message Attributes, they are:

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.

Based on Queue Attributes, they are:

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 message.
QUE_Q_LPQ	The candidate message from the queue having the lowest priority message.
QUE_Q_EAQ	The candidate message from the queue having the earliest arrived message.
QUE_Q_LAQ	The candidate message from the queue having the latest arrived message.
QUE_Q_ANY	The first candidate message.

Thus, for example, the retrieval expression:

```
QUE_Q_HP { QUE_M_EA(a), QUE_M_EA(b) }
```

can be used to summarize the following retrieval operation: "Compare the oldest (Earliest Arrived) message on queue **a** with the oldest message on queue **b** and return the one with the highest priority."

Similarly:

```
QUE_Q_EA { QUE_M_HP(x), QUE_M_HP(y), QUE_M_HP(z) }
```

returns the oldest of the highest priority messages resident on queues x, y and z.

Finally, consider a retrieval example having a slightly different slant:

```
QUE_Q_LNQ { QUE_M_HP(a), QUE_M_HP(b), QUE_M_HP(c) }
```

First, the highest priority message from each of the three respective queues **a**, **b** and **c** are designated as competing candidate messages. The returned message is then selected to be the one residing on the longest of the three queues.

Possible "Message Select Codes" and their interpretations are:

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 greater than or equal to 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 less than or equal to 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)	The first message on queue Q with a value equal to sequence number *seqn*.
QUE_M_SEQGE (Q, seqn)	The first message on queue Q with a value greater than or equal to sequence number *seqn*.
QUE_M_SEQLE(Q, seqn)	The first message on queue Q with a value less than or equal to sequence number *seqn*.
QUE_M_SEQGT(Q, seqn)	The first message on queue Q with a value greater than sequence number *seqn*.
QUE_M_SEQLT(Q, seqn)	The first message on queue Q with a value less than sequence number *seqn*.

The use of "Queue Select Codes" and "Message Select Codes" to define message transfer operations offers the flexibility and functionality needed by most queue intensive applications.

A presentation on the optimum usage of MSC and QSC codes for message dispatch and retrieval operations is included in the Advanced Topics chapter of this guide.

2.2 QueSys Configuration

The QueSys section of an X•IPC instance configuration file describes the composition and capacity of the instance's QueSys.

Seven parameters must be set within the QueSys section of the instance configuration. Parameter tables with default values are provided at the beginning of the companion Reference Manual. Additional operating system specific parameters (if required) are described in the relevant Platform Notes.

The configuration parameters are:

- **MAX_QUEUES**, The maximum number of concurrent queues. Should be set based on the requirements of the programs using the instance.
- **MAX_USERS**, The maximum number of concurrent users. 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.
- **MAX_NODES**, The maximum number of nodes. QueSys nodes are used internally for tracking users that block on QueSys operations. As with SemSys, there is no firm rule for calculating a value for **MAX_NODES**. It depends largely on the nature of the programs that will use the instance. A conservative estimate to start with can be calculated from the following formula:

  MAX_NODES = MAX_QUEUES+(MAX_USERS*3)+(MAX_USERS* MAX_QUEUES)
- **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 starting formula for **MAX_HEADERS** is:

  MAX_HEADERS = MAX_QUEUES + (MAX_QUEUES * *AverageQueueLength*)

  where:

  AverageQueueLength is the expected average queue length (in terms of messages) within the instance.

- **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 formula for **SIZE_MSGPOOL** is:

SIZE_MSGPOOL=(**MAX_QUEUES***AverageQueueLength*)*(*AverageMessageSize*+16)

where:

AverageQueueLength is as defined above.

AverageMessageSize is the expected average message size occurring within the instance.

SIZE_MSGPOOL is expressed in terms of K-Bytes. As such the calculated value should be rounded up to the next K-Byte multiple. (For example, if the calculation comes to 1948 bytes, then 2 K-Bytes should be specified).

- **SIZE_MSGTICK**, 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 formula for a good starting value for **SIZE_MSGTICK** is:

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.

- **SIZE_SPLTICK**, The spool tick file size limit (K-Bytes). Defines the file size limit used in the course of 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. On the other hand, 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, at a minimum, **SIZE_SPLTICK** must be 32 bytes larger than the largest message to be spooled by any queue in the instance.

Example:

Consider the QueSys configuration below for an *X•IPC* instance that will support a high performance transaction processing application.

Assumptions:

1. There will be between 5 and 10 users and/or asynchronous QueSys operations at any one point in time.
2. There will be between 10 and 15 queues active at any one time.
3. The average queue length is expected to range between 25 and 30 messages.

4. The expected average message size is 32 bytes.
5. It is estimated that 25% of the messages will be less than 21 bytes in size, and that 75% of the messages will exceed 21 bytes in length. Thus 21 bytes is the estimated *25PercentileMessageSize*.
6. Spool files must not exceed 128 K-Bytes. The largest message to be spooled will not exceed 1024 bytes.

Then:

MAX_USERS can be safely set at 10. Little space is required for configuring extra users, so it pays to play it safe.

MAX_QUEUES can be set at 15. The **MAX_USERS** reasoning is valid here as well.

MAX_NODES follows then as: 15 + (10 * 3) + (10 * 15) = 195.

MAX_HEADERS would be calculated as: 15 + (15 * 30) = 465.

SIZE_MSGPOOL would be calculated as: (15 * 30) * (32 + 16) = 21,600. The number 22 can be used since 21,600 < 22 K-Bytes

SIZE_MSGTICK would be set to 24 bytes since it is the next multiple of 4, after 21.

SIZE_SPLTICK can be set at 128 K-Bytes.

```
#==================================================================
#
# File: /projects/local/tpsys.cfg
# Created: May 31, 2001
#
#------------------------------------------------------------------
#
# This XIPC instance supports a high-performance
# transaction processing application.
# Note: The instance is defined so that it only
# supports XIPC QueSys queues. The SemSys, MemSys
# and MomSys subsystems are defined as NULL.
#
#------------------------------------------------------------------


[QUESYS]
MAX_USERS       10
MAX_QUEUES      15
MAX_NODES       195
MAX_HEADERS     465
SIZE_MSGPOOL    22
SIZE_MSGTICK    24
SIZE_SPLTICK    128

#==================================================================
```

A further note about QueSys configuration: the above formulae and rules generally produce acceptable parameter values. The values should, however, be adjusted as necessary based on empirical observations using the QueSys monitor.

2.2.1 QueSys Configuration

The QueSys section of an X•IPC instance configuration file describes the composition and capacity of the instance's QueSys.
Seven parameters must be set within the QueSys section of the instance configuration. Parameter tables with default values are provided at the beginning of the companion Reference Manual. Additional operating system specific parameters (if required) are described in the relevant Platform Notes.

The configuration parameters are:

- **MAX_QUEUES**, The maximum number of concurrent queues. Should be set based on the requirements of the programs using the instance.
- **MAX_USERS**, The maximum number of concurrent users. 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.
- **MAX_NODES**, The maximum number of nodes. QueSys nodes are used internally for tracking users that block on QueSys operations. As with SemSys, there is no firm rule for calculating a value for **MAX_NODES**. It depends largely on the nature of the programs that will use the instance. A conservative estimate to start with can be calculated from the following formula:

  MAX_NODES = MAX_QUEUES+(MAX_USERS*3)+(MAX_USERS*MAX_QUEUES)
- **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 starting formula for **MAX_HEADERS** is:

  MAX_HEADERS = MAX_QUEUES + (MAX_QUEUES * *AverageQueueLength*)

  where:

  AverageQueueLength is the expected average queue length (in terms of messages) within the instance.
- **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 formula for **SIZE_MSGPOOL** is:

SIZE_MSGPOOL=(**MAX_QUEUES***_AverageQueueLength_)*(_AverageMessageSize_+16)

  where:

  AverageQueueLength is as defined above.

  AverageMessageSize is the expected average message size occurring within the instance.

  SIZE_MSGPOOL is expressed in terms of K-Bytes. As such the calculated value should be rounded up to the next K-Byte multiple. (For example, if the calculation comes to 1948 bytes, then 2 K-Bytes should be specified).
- **SIZE_MSGTICK**, 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 formula for a good starting value for **SIZE_MSGTICK** is:

  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.

- **SIZE_SPLTICK**, The spool tick file size limit (K-Bytes). Defines the file size limit used in the course of 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. On the other hand, 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, at a minimum, **SIZE_SPLTICK** must be 32 bytes larger than the largest message to be spooled by any queue in the instance.

Example:

Consider the QueSys configuration below for an X•IPC instance that will support a high performance transaction processing application.

Assumptions:

1. There will be between 5 and 10 users and/or asynchronous QueSys operations at any one point in time.
2. There will be between 10 and 15 queues active at any one time.
3. The average queue length is expected to range between 25 and 30 messages.
4. The expected average message size is 32 bytes.
5. It is estimated that 25% of the messages will be less than 21 bytes in size, and that 75% of the messages will exceed 21 bytes in length. Thus 21 bytes is the estimated *25PercentileMessageSize*.
6. Spool files must not exceed 128 K-Bytes. The largest message to be spooled will not exceed 1024 bytes.

Then:

MAX_USERS can be safely set at 10. Little space is required for configuring extra users, so it pays to play it safe.

MAX_QUEUES can be set at 15. The **MAX_USERS** reasoning is valid here as well.

MAX_NODES follows then as: 15 + (10 * 3) + (10 * 15) = 195.

MAX_HEADERS would be calculated as: 15 + (15 * 30) = 465.

SIZE_MSGPOOL would be calculated as: (15 * 30) * (32 + 16) = 21,600. The number 22 can be used since 21,600 < 22 K-Bytes

SIZE_MSGTICK would be set to 24 bytes since it is the next multiple of 4, after 21.

SIZE_SPLTICK can be set at 128 K-Bytes.

```
#=================================================================
#
# File: /projects/local/tpsys.cfg
# Created: May 31, 2001
```

```
#
#-----------------------------------------------------------------
#
# This XIPC instance supports a high-performance
# transaction processing application.
# Note: The instance is defined so that it only
# supports XIPC QueSys queues. The SemSys, MemSys
# and MomSys subsystems are defined as NULL.
#
#-----------------------------------------------------------------


[QUESYS]
MAX_USERS       10
MAX_QUEUES      15
MAX_NODES       195
MAX_HEADERS     465
SIZE_MSGPOOL    22
SIZE_MSGTICK    24
SIZE_SPLTICK    128

#=================================================================
```

A further note about QueSys configuration: the above formulae and rules generally produce acceptable parameter values. The values should, however, be adjusted as necessary based on empirical observations using the QueSys monitor.

2.3 QueSys Functions

2.3.1 QueCreate() - Creating a New Queue

The first step toward using a QueSys message queue within an instance is to create the queue.
QueCreate() takes three arguments:

- ☐ The name of the new queue.
- ☐ A value specifying the message capacity of the queue (if any).
- ☐ A value specifying the byte capacity of the queue (if any).

QueCreate() returns the "queue id" (Qid) of the newly created queue. This value is used as the queue's "handle" in all subsequent QueSys function calls that refer to the queue.

Example:

```
Qid = QueCreate("InputQueue", 50, 1024L);
```

In the above example, the calling user attempts to create a new message queue with the name "InputQueue." The new queue will have a capacity limit of 50 messages and 1024 bytes. The queue is considered full when one or the other of these limits is reached.

Example:

```
Qid = QueCreate("LimitedBytes", QUE_NOLIMIT, 32768L);
```

In this example, the calling process is creating a message queue with the name "LimitedBytes." a byte capacity limit of 32,768 (32 K-Bytes) and an unlimited message capacity.
Having an unlimited message capacity means that the queue can hold as many messages as necessary, provided the byte capacity (32 K-Bytes) is not violated. This is very useful for limiting queues whose message sizes are unknown. In this example, any combination

of messages (large, small or a mixture) that fills the queue with 32 K-Bytes will render the queue full.

Example:

```
Qid = QueCreate("LimitedMsgs", 100, QUE_NOLIMIT);
```

Here, the calling user creates a message queue with the name "LimitedMsgs." The queue is created with a capacity limit of 100 messages and an unlimited byte capacity.
This type of queue would be useful for implementing a priority queue that had to accept a limited number of messages (in this case 100) regardless of their sizes.

Example:

```
Qid = QueCreate("Very_Big_Queue", QUE_NOLIMIT, QUE_NOLIMIT);
```

In this example, a queue is created having unlimited message and byte capacities. Such a queue might be used for collecting incoming messages from a producer that could not be blocked and for which spooling was *not* appropriate.
Such a queue would continue to accept messages until some QueSys configuration limit was reached. This may be a lack of free space in the message text pool, or a shortage of available message headers.
Creating a queue with unlimited capacity values should be restricted to special situations. Such uncontrolled queues can wreak havoc if used inappropriately.
A further note regarding queue creation: duplicate queue names are not allowed within an instance.
Specifying `QUE_PRIVATE` as the name of the new queue creates a queue that is inaccessible via QueAccess(), effectively making its 'Qid' private to the creating program. Of course, the creating program can pass the 'Qid' to others if it so wishes. The advantage of using `QUE_PRIVATE` as a name is that it is guaranteed not to conflict with any queue name currently in the instance.

2.3.2 QueAccess() - Accessing an Existing Queue

Once a queue has been created, other users can access it (i.e., its Qid) using QueAccess(). QueAccess() takes one argument:

- ☐ The name of an existing message queue.

QueAccess() returns the "queue id" (Qid) of the desired queue. This value is used as the message queue's "handle" in all subsequent QueSys function calls that refer to the queue.

Examples:

```
Qid  = QueAccess("InputQueue");
Qid  = QueAccess("LimitedBytes");
Qid  = QueAccess("LimitedMsgs");
```

The above examples access three of the queues created in the previous section.

2.3.3 QueWrite() - Writing Message Text to the Text Pool

To send a message onto a queue using the header/text approach outlined earlier, you must first write the message's text to the QueSys message text pool via QueWrite(). This will usually be followed by a call to QuePut() to place the corresponding message header onto the desired queue.

Both of these steps can be avoided by using the QueSend() function. It performs the text write and the header placement in one operation. QueSend() is described later. In addition to writing text to the message text pool, QueWrite() also creates a message header corresponding to the written text. This header is what actually moves about on QueSys queues throughout the duration of the message's existence. QueWrite() sets the created message text pool block's reference count to one (1) when it creates a new text block.

QueWrite() takes four arguments:

- A pointer to an empty message header.
- A pointer to the text to be written.
- The size of the message text (in bytes).
- A blocking option code in case the operation needs to block.

The empty message header passed (indirectly) to QueWrite() is returned with appropriate values. It can be subsequently placed onto a queue using QuePut().
Note that `MSGHDR` is a datatype provided by *X•IPC* QueSys for working with message headers.

Example:

```
/*
 * Write a "hello world" message text to the message text pool.
 * In the process create a MSGHDR variable corresponding to the
 * written text - ready for placement onto a queue.
 */

MSGHDR MessageHeader;

RetCode = QueWrite(&MessageHeader, "hello world", 11L, QUE_WAIT);

if (RetCode >= 0)
    RetCode = QuePut(&MessageHeader, ... );
```

This example demonstrates an important QueSys capability: being able to block when the text pool is full until the required text space becomes available. Specifying `QUE_WAIT` directs QueWrite() to block indefinitely if the required text space is not currently available. The user unblocks after the write finally succeeds.
Blocking at the text pool level is not possible with most other queuing facilities.

Example:

```
/*
 * Write "hello world" message text to the text pool.
 * Block for 30 seconds if the message text pool is full.
 */

MSGHDR MessageHeader;

RetCode = QueWrite(&MessageHeader,
                   "hello world",
                   11L,
                   QUE_TIMEOUT(30));

if (RetCode >= 0)
    /* Write succeeded */

else
if (RetCode == QUE_ER_TIMEOUT)
    /* Handle timeout */
```

In this example, the QueWrite() function is instructed to block no more than 30 seconds while waiting for text space.
Specifying `QUE_NOWAIT` as the blocking option would cause QueWrite() to return immediately with an error code (`RetCode == QUE_ER_NOWAIT`) if the required text space was not immediately available.
The above examples demonstrate QueWrite() using synchronous blocking options. Asynchronous blocking is also possible by specifying one of the three asynchronous blocking options. Refer to the Advanced Topics chapter for a detailed description of the asynchronous blocking options.

2.3.4 QueRead() - Reading Message Text from the Text Pool

QueRead() reads message text out of the message text pool and copies it to a user-specified location. QueRead() uses a message header argument for identifying the message text to be read. This message header will, in most cases, have just been attained from a QueSys queue via a QueGet() operation. QueRead() accesses the message's text using its header. QueRead() decrements the text pool block's reference count by one (1); if the count equals zero, then it will release the text block.

QueRead() takes three arguments:

- A message header (via a pointer).
- A pointer to a buffer to receive the message text.
- A long integer specifying the maximum size text to copy into the buffer (usually the size of the buffer).

QueRead(), when successful, returns the number of bytes read from the text pool.
If the stipulated buffer size limit is greater than or equal to the actual message text size, then the QueRead() operation will succeed and the entire message text will be copied.
If, however, the buffer size limit is less than the message text size, then truncation is possible. Truncation will occur if and only if specified by the user via the `QUE_TRUNCATE` macro as demonstrated below. Otherwise, an error code (`RetCode == QUE_ER_TOOBIG`) is returned and the QueRead() operation will fail.

Example:

```
/*
 * Retrieve a message header from a message queue.
 * Then, read the message's text into 'Buff'.
 * In this example, the QueRead will fail if the
 * message text exceeds 100 bytes in length.
 */

MSGHDR MessageHeader;
CHAR Buff[100];
XINT MsgLen;

RetCode = QueGet(&MessageHeader, ...);

MsgLen = QueRead(&MessageHeader, Buff, 100L);

if (MsgLen == (XINT)QUE_ER_TOOBIG)
    /* Message text exceeds 100 bytes. */
```

Example:

```
/*
 * Same example as above ...
 * In this version, the QueRead will truncate the message text
 * if its length exceeds 100  bytes. The first 100
 * bytes will be copied, the remaining text bytes are lost.
 */

MSGHDR MessageHeader;
CHAR Buff[100];
XINT MsgLen;

RetCode = QueGet(&MessageHeader, ...);

MsgLen = QueRead(&MessageHeader, Buff, QUE_TRUNCATE(100L));
```

You can avoid using two separate functions, QueGet() and QueRead(), to retrieve a message by using the QueReceive() function. QueReceive() performs the QueGet() and QueRead() as a single operation (described later in this Guide).

2.3.5 QueListXxx() - Queue List Manipulation Functions

QueSys operations use lists of Qids to dispatch and retrieve messages. Message transfer involving only one queue is accomplished using a single-element list.
A list of Qids is referred to as a QidList. A `QIDLIST` data type is defined for creating and working with QidLists. Functions expecting a QidList as one of their arguments take a `QIDLIST` data type for this purpose.
There are two functions for building QidLists: QueList() and QueListBuild(). The difference between the two is that QueList() builds its QidList in its internal static area, thus making the returned QidList safe for only one usage. QueListBuild(), on the other hand, takes a `QIDLIST` variable as its first argument. QueListBuild() builds its QidList in this user-provided space. This QidList can safely be reused by the programmer. Apart from this, the two functions are otherwise identical. Both functions require `QUE_EOL` as their last argument.

Two additonal functions, QueListAdd() and QueListRemove(), allow for updating QidLists dynamically, and another function, QueListCount(), allows determination of the number of elements in a QidList..
QueListAdd() is provided to allow the programmer to add QidList elements to an existing QidList (i.e., one that has been created by QueListBuild()). This is a common requirement in situations where the needed QidList must be built dynamically, based on certain run-time conditions.
QueListRemove() is provided to allow the programmer to remove QidList elements from an existing QidList when necessary.
The calling sequence for QueListAdd() and QueListRemove() is identical to that of QueListBuild(). These too expect a user-provided QidList as their first argument. The listed QidList elements are added to or removed from that QidList.

Example:

```
/*
 * The following code constructs two identical QidLists.
 * QidList1 is constructed using a single function call.
 * QidList2 is built incrementally, one Qid at a time.
 */

QIDLIST QidList1, QidList2;

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

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

Example:

```
/*
 * The following code waits to receive a message from any of the
 * queues in QidList, handling the possibility that any number of
 * those queues might be destroyed while waiting.
 */

do {
   RetCode = QueReceive(QUE_Q_EA, QidList, Buf, Len, NULL, &RetQid,
QUE_WAIT);
   if (RetCode == QUE_ER_DESTROYED)
   {
      QueListRemove(QidList, RetQid, QUE_EOL);
      if (QueListCount(QidList)==0)
         break;
      else
         continue;
   }
} while(RetCode == QUE_ER_DESTROYED);
```

2.3.5.1 Message Dispatch QidLists

When preparing a QidList for a QuePut() or a QueSend() operation, the elements in the list are the Qids of the queues to be considered as the target of the QuePut() or the QueSend() call. The actual target selection is based on the Queue Select Code argument of the function call. This process was described earlier.

Example:

```
/*
 * Construct a QidList and then use it to send a message to the
 * shortest of the listed target queues.
 */

QIDLIST QidList;

QueListBuild(QidList, QidA, QidB, QidC, QUE_EOL);

RetCode = QueSend(QUE_Q_SHQ, QidList, ... );
```

The above QueSend() call would send its message onto the shortest of the three queues represented by QidA, QidB and QidC. In fact, the very same QidList could be reused to send a message to the longest queue of the list by simply specifying the QUE_Q_LNQ Queue Select Code.

Example:

```
/*
 * Use the previously constructed QidList to send a message
 * onto the longest queue of the list: QidA, QidB, QidC.
 */

RetCode = QueSend(QUE_Q_LNQ, QidList, ... );
```

QueList() returns a pointer to the QidList that it constructs internally. The call to QueList() is therefore normally embedded directly as the argument for the dispatch function being invoked.

Example:

```
/*
 * This QueSend call is equivalent to the one shown in the
 * previous example.
 */

RetCode = QueSend(QUE_Q_LNQ,
                  QueList(QidA, QidB, QidC, QUE_EOL),
                  ... );
```

The usage of QueList() and QueListBuild() within the context of QuePut() operations is identical to the QueSend() examples shown above.

2.3.5.2 Message Retrieval QidLists

Building a QidList for a QueGet() or a QueReceive() operation is slightly more involved than preparing one for dispatch. In this case the QidList serves two purposes:

- It presents a list of source Qids from which to consider retrieving a message.
- It identifies a "candidate message" for each of the listed source Qids.

Each element of a message retrieval QidList is usually a Message Select Code (MSC) applied to a Qid. Each MSC defines the criteria to be used for choosing its queue's candidate message. The following examples clarify this point.

Example:

```
/*
 * Build a QidList that will designate the highest priority
 * message from each of the queues QidA, QidB and QidC as
 * the three queues' respective candidate messages. Then,
 * retrieve the oldest (earliest arrived) of these messages.
 */

QIDLIST QidList;
XINT MsgLen;

QueListBuild(QidList,
             QUE_M_HP(QidA), QUE_M_HP(QidB), QUE_M_HP(QidC),
             QUE_EOL);

MsgLen = QueReceive(QUE_Q_EA, QidList, ... );
```

The net effect is the retrieval of the oldest of the high priority messages from the three queues: QidA, QidB and QidC. (Note that specification of the lowest priority messages would have resulted in the retrieval of the newest of those messages.)
Note how the QUE_M_HP Message Select Code macro is applied to each of the queues in the list to identify each queue's candidate message.
Here, too, QidLists created using QueListBuild() can be reused for other purposes. For example, you can use the same QidList with the QUE_Q_LNQ (longest queue) Queue Select Code to retrieve the highest priority message from the longest of the three queues: QidA, QidB and QidC. More precisely, it selects the candidate message from the longest of the three queues.
Example:

```
/*
 * Reuse the QidList built earlier. This time retrieve the
 * message residing on the longest of the three queues.
 */

MsgLen = QueReceive(QUE_Q_LNQ, QidList, ... );
```

The possibilities are virtually endless. Refer to the section in the Advanced Topics chapter on "Using Message Select Codes and Queue Select Codes" for a more complete description of this facility and its options.
QidLists must not exceed QUE_MAX_QIDLIST elements. This is usually not a great concern since QUE_MAX_QIDLIST is currently defined as 32.

QidList Simplification

QidList simplification during message retrieval operations is possible in certain cases. A simplified QidList produces the same retrieval operation outcome as would have resulted using the original QidList. The following example demonstrates this concept.

Example:

```
/*
 * Two QueReceive operations that are equivalent.
 */

QueReceive(QUE_Q_HP,
           QueList(QUE_M_HP(x), QUE_M_EA(y), QUE_M_HP(z),
                   QUE_EOL),
           ... );

QueReceive(QUE_Q_HP,
           QueList(x, QUE_M_EA(y), z, QUE_EOL),
           ... );
```

Both QueReceive() calls 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**.

Both QueReceive() calls retrieve the candidate message having the highest priority.
The following simplification has occurred: The first and third Qids of the simplified QidList in the second example above lack Message Select Codes. As a result they "inherit" the criteria of the operation's Queue Select Code ("high priority").

The simplification rule can be formulated as:

> "Whenever a message retrieval QidList has a Qid entry for which no Message Select Code is provided, the retrieval operation's Queue Select Code criteria is employed as the Message Select Code for that queue."

QidList simplification is often quite useful, as shown in the following examples:
Example:

```
/*
 * Two more QueReceive operations that are equivalent.
 */

QueReceive(QUE_Q_HP,
           QueList(QUE_M_HP(q),
                   QUE_M_HP(r),
                   QUE_M_HP(s),
                   QUE_EOL),
           ... );

QueReceive(QUE_Q_HP, QueList(q, r, s, QUE_EOL), ...);
```

Both retrieval operations return the overall highest priority message from the three queues: **q**, **r** and **s**.
Both operations first designate a candidate message from each of the three queues: **q**, **r** and **s**. They are the highest priority message of each of the queues.
The three candidate messages are then compared and the highest priority message of the three candidates is chosen for retrieval.
QidList simplification makes complex retrieval operations easier to express and understand. In fact, if a retrieval QidList is completely simplified so that it no longer contains any Message Select Codes, then that QidList can be used in message dispatch operations as well.
Example:

```
/*
 * Create a QidList that can be used for both
 * dispatch and retrieval operations.
 */

QIDLIST QidList;
XINT MsgLen; QueListBuild(QidList, qa, qb, qc, QUE_EOL);

/*
 * Send a message onto the queue having the
 * earliest arrived(oldest) message.
 */

RetCode = QueSend(QUE_Q_EAQ,
                  QidList, ... );

/*
 * Now use the same QidList to receive the lowest priority
 * message of the same three queues.
 */

MsgLen = QueReceive(QUE_Q_LP, QidList, ... );
```

Priority Specification During Message Retrieval

Message retrieval based on message priorities is accomplished using the priority-related Message Select Codes. A number of the Message Select Codes deal with priorities. Priority values, conditions or ranges can be specified.

Example:

```
QIDLIST QidList;
XINT MsgLen;

QueListBuild(QidList, QUE_M_PREQ(a, 100), QUE_M_PRLT(b, 50),
             QUE_EOL);

MsgLen = QueReceive(QUE_Q_EA, QidList, ... );
```

In the above example, a QidList is built designating 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**.
The QueReceive() call then returns the earliest arrived (oldest) of these two candidate messages.
Similarly:

```
QIDLIST QidList;
XINT MsgLen;

QueListBuild(QidList,
             QUE_M_PRRNG(a, 100, 200),
             QUE_M_PRRNG(b, 100, 200),
             QUE_EOL);

MsgLen = QueReceive(QUE_Q_LNQ, QidList, ... );
```

In this example, the QueReceive() operation accesses 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.

Similar Message Select Codes are provided for selecting message via their sequence number. A complete list of Queue and MessageSelect Codes is found in the Advanced Topics chapter of this book.

2.3.6 QuePut() - Putting a Message Header onto a Queue

The QuePut() function is used to place a message header onto a queue. The message header involved may have just been created via QueWrite(), or it may have been recently removed from one queue for re-routing onto another queue.

QuePut() takes six arguments:

- A pointer to the message header being dispatched.
- A Queue Select Code for choosing a target queue.
- A QidList holding a list of possible target Qids.
- A priority value to be assigned to the message.
- A pointer to a Qid variable that gets assigned the actual target Qid chosen by QuePut(). (This pointer can be `NULL` if no return value is desired.)
- A blocking option code in case the operation needs to block.

Example:

```
/*
 * Create a message and place it onto the "OutQueue" queue.
 * Assign the dispatched message a priority of 99.
 * Warning: No error checking is done in this example.
 */

XINT Qid;
XINT RetQid;
MSGHDR MsgHdr;

Qid = QueAccess("OutQueue");

QueWrite(&MsgHdr, "hello world", 11L, QUE_WAIT);

QuePut(&MsgHdr, QUE_Q_ANY, QueList(Qid, QUE_EOL), 99L,
       &RetQid, QUE_WAIT);
```

This is an example of a QidList with a single Qid element. The `QUE_Q_ANY` Queue Select Code finds the first queue in the QidList that has space for the message and places the message there. In this case, the first queue is the only queue. Remember that QuePut() does not touch a message's text. It simply places a given message header onto the queue whose identity is determined by the Queue Select Code, the QidList arguments and the traffic capacity of the listed queues. We will see that the above example could have been coded more concisely using the QueSend() function call.

Any of the *X•IPC* blocking options can be specified for QuePut(). QuePut() will block if all of the listed queues are full, making them unable to accept the message being dispatched. The blocked operation will then succeed when any of the full queues is no longer at capacity, usually when one or more messages have been removed.

QuePut() will not block if any of the listed queues is actively spooling (i.e., has its spooling on). In the event that all listed queues are full, then the message will be spooled out to the selected queue's spool. A detailed description of queue overflow spooling is presented later in this chapter.

QuePut() returns the identity of the queue used, by assigning its Qid to "RetQid." "RetQid" is also used to identify an invalid Qid, if one is encountered.
Example:

```
/*
 * Send a message (header) onto the queue having the oldest
 * (i.e. earliest arrived) message. Wait up to 45 seconds for
 * QuePut to succeed. Then report which queue was used. The
 * dispatched message is assigned a priority of 2500.
 */

QIDLIST QidList;
XINT QidSent;

QueListBuild(QidList, QidA, QidB, QidC, QUE_EOL);

if (QuePut(&MsgHdr, QUE_Q_EA, QidList, 2500L, &QidSent,
        QUE_TIMEOUT(45)) >= 0)
        printf("Message was placed onto Qid = %d\n", QidSent);
```

The above examples demonstrate QuePut() using synchronous blocking options. Asynchronous blocking is also possible by specifying one of the three asynchronous blocking options. Refer to the Advanced Topics chapter for a detailed description of the asynchronous blocking options.
The QUE_REPLICATE 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*.
It is also possible for a message header copy what was retrieved from a queue using QueGet's QUE_NOREMOVE option to be placed onto a message queue using QuePut().
Example:

```
/*
 * Same as prior example, but this time have message copies sent
 * to *** ALL *** users currently waiting for such a message.
 */

. . .
. . .
QuePut (&MsgHdr, QUE_Q_ANY, QueList(Qid, QUE_EOL), 99L,
        &RetQid, QUE_REPLICATE );
```

Note that QUE_REPLICATE is specified in place of any other X•IPC blocking option. The QuePut() call will never block. Message (header) copies are sent to all (zero or more) users waiting for the message regardless as to the current number of messages on the specified queue.
For additional details, see the Advanced Topics section, "QueSys Message Multicasing."

2.3.7 QueGet() - Getting a Message Header From a Queue

You can retrieve a message header from a queue by using the QueGet() function call.

A retrieved message header can then be used in a variety of ways, including:

- To examine the message's text via QuePointer() or QueCopy().
- To browse the queue's messages, relative to the retrieved header.
- To read the message's text via QueRead(), from the text pool into user memory space.
- The message header can also be placed onto another queue via QuePut().

QueGet() takes six arguments:

- A pointer to an empty message header that gets set with the retrieved message's header data.
- A Queue Select Code for choosing one of the candidate messages.
- A QidList identifying candidates messages for each of the listed queues.
- A pointer to a variable that is assigned the retrieved message's priority.
- A pointer to a Qid variable that is set (on return) to the Qid of the retrieved message's queue by QueGet(). (This pointer can be `NULL` if no return value is desired.)
- An optional `QUE_NOREMOVE` option flag ORed with a blocking option code, specifying the action to be taken in case the operation needs to block.

Keep in mind that QueGet() does not touch a message's text. It simply gets a message header from the queue, whose identity is determined by the Queue Select Code, the QidList and the messages currently on the listed queues.

The retrieved message header is copied into the user-provided message header variable. In the default case (i.e., `QUE_NOREMOVE` is *not* specified), QueGet() *removes* the retrieved message header from the queue that it was on. Alternatively, by specifying the `QUE_NOREMOVE` option flag, you can direct QueGet() to *leave* the accessed message header on the queue and to return a copy of it to the calling program, a copy that is itself a fully functional message header. In such a case, the returned header copy can be placed on another queue and used as a reference point for a subsequent QueBrowse() call. Or, it can be used to later remove the actual header from the queue via QueRemove().

In either case, QueGet() leaves the message's text untouched in the message text pool.

Here again, "RetQid" identifies the Qid retrieved. "RetQid" is also used to identify an invalid Qid, if one is encountered.

Reacting to the retrieved message may depend on the queue from which it was retrieved. The next example demonstrates this point. In some cases, it might be appropriate to read in the message's text for processing; in other cases, re-routing it onto another queue may be called for.

Note that a message's priority is set at QuePut() time and can therefore be changed (as in the next example) between legs of a multi-queue journey.

Example:

```
/*
 * Retrieve the highest priority message in the range
 * [200,400] from across the three queues identified by
 * QidA, QidB and QidC. React to message based on its
 * source Qid.
 */

QIDLIST QidList;
MSGHDR MsgHdr;
XINT Priority;
XINT RetQid;
XINT NextQid;
CHAR Buff[100];
XINT Length;

QueListBuild(QidList,
             QUE_M_PRRNG(QidA, 200L, 400L),
             QUE_M_PRRNG(QidB, 200L, 400L),
             QUE_M_PRRNG(QidC, 200L, 400L),
             QUE_EOL);
RetCode = QueGet(&MsgHdr, QUE_Q_HP, QidList, &Priority,
                 &RetQid, QUE_WAIT);

if (RetCode >= 0)
{
    /*
     * QueGet succeeded, 'RetQid' was set with Qid of the
     * source queue, and 'Priority' was set with the gotten
     * message's priority.
     */

    if (RetQid == QidA  ||  RetQid == QidB)
    {
        /*
         * QidA and QidB are treated identically. Read the
         * message text out of the message text pool into
         * user space, and process it.
         */

        Length = QueRead(&MsgHdr, Buff, 100L);
        ...
    }




    else
    {
        /*
         * Re-route QidC message onto another queue.
         * Give it a higher priority.
         */

        RetCode = QuePut(&MsgHdr, QUE_Q_ANY, QueList(NextQid, QUE_EOL),
                         Priority + 100L, &RetQid, QUE_WAIT);
        ...
    }
}
```

A QidList can include multiple MSCs for a single queue. This is extremely useful if it is necessary to access messages from non-contiguous sections of a priority range.
We will see shortly that basic message retrieval operations can be coded more concisely using the QueReceive() function call. QueGet(), however, provides maximum flexibility for dealing with message headers.
Any of the *X•IPC* blocking options can be specified for QueGet(). QueGet() will block if all of the listed queues are empty or if they do not currently hold a desired message (as specified by the Message Select Codes).
The blocked operation will succeed when one of the queues receives a message sought by the blocked QueGet() operation.

Example:

```
/*
 * Access the oldest message within the ranges [1,10] and
 * [90,100]. Note: Only one queue is involved.
 */

QIDLIST QidList;
MSGHDR MsgHdr;
XINT qa;
XINT Priority;
XINT RetQid;
XINT RetCode;

QueListBuild(QidList,
             QUE_M_PRRNG(qa, 1L, 10L),
             QUE_M_PRRNG(qa, 90L, 100L),
             QUE_EOL);

RetCode = QueGet(&MsgHdr, QUE_Q_EA, QidList, &Priority,
                 &RetQid, QUE_WAIT);

if (RetCode >= 0)    /* QueGet succeeded. */
{
    if (Priority >=1L && Priority <=10L)
    {
        ...
    }
    else /* Priority >=90L && Priority <= 100L */
    {
        ...
    }
}
```

As was stated above, QueGet() can be used with the QUE_NOREMOVE option to access a copy of a message header, without removing the actual header from the queue.

Example:

```
/*
 * Access an image of the highest priority message
 * header on queue Qid.
 */

RetCode = QueGet(&MsgHdr,
                 QUE_Q_HP,
                 QueList(Qid, QUE_EOL),
                 &Priority,
                 &RetQid,
                 QUE_NOREMOVE | QUE_WAIT );
```

Note that the QUE_NOREMOVE option flag, when specified, *must* precede whatever blocking option is designated; this is because QUE_WAIT expands to several arguments, the first of which can be ORed with MOM_NOREMOVE. The returned message header is a copy of the actual message header that is left on the queue.
Such a header can be used for a subsequent QueBrowse() operation. QueBrowse() will only succeed if the message header it is passed references a header still on a queue.
By contrast, a message header copy *cannot* be used within a QueUnget() operation. This function will only succeed when passed a message header that has actually been dequeued.
The above examples demonstrate QueGet() using synchronous blocking options. Asynchronous blocking is also possible by specifying one of the three asynchronous blocking options. Refer to the Advanced Topics chapter for a detailed description of the asynchronous blocking options.

2.3.8 QueRemove() - Remove a Message Header from a Queue

QueRemove() dequeues the message header identified by the message header copy it is passed. The message header parameter must be a copy of a message header that still resides on a queue.

QueRemove() takes one argument:

- ☐ The message header that has not been dequeued.

The message header copy may have been accessed through a QueGet() operation where QUE_NOREMOVE was specified, or via a QueBrowse() operation.

Example:

```
/*
 * Remove the message header referenced by
 * the message header retrieved in the
 * previous example.
 */

RetCode = QueRemove ( &MsgHdr );
```

A message can be accessed in two steps, using QueGet() with the QUE_NOREMOVE option specified followed by a call to QueRemove(). This is equivalent to using QueGet() *without* the QUE_NOREMOVE option. The removed header may be placed onto another queue via QuePut() or its text can be read via a call to QueRead().

2.3.9 QueSend() - Sending a Message onto a Queue

As was indicated earlier, it is often not necessary to use QueWrite() followed by QuePut() to dispatch a message. A single function, QueSend(), can be used instead when the two steps of control available using QueWrite() and QuePut() are not required. "QueWrite() + QuePut() = QueSend()" summarizes the functionality of the QueSend() function.

Advantages of using QueSend() include:

- No need to manage message headers.
- No need to interact directly with the message text pool.
- Generally slightly better performance than "QueWrite() + QuePut()".

Disadvantages include:

- An inability to manipulate message headers independent of their associated text segments; inter-queue routing using QueSend() is therefore inefficient.
- An inability to specify different blocking options for the "text write" and "message put" components of the QueSend() operation.

QueSend() takes seven arguments:

- A Queue Select Code for choosing a target queue.
- A QidList holding a list of possible target Qids.
- A pointer to the text to be written.
- The size of the message text (in bytes).
- A priority value to be assigned to the message.
- A pointer to a Qid variable that gets set by QueSend() to the actual target Qid chosen. (This pointer can be `NULL` if no return value is desired.)
- A blocking option code in case the operation needs to block.

Example:

```
/*
 * Send a "hello world" message onto the queue represented by
 * Qid "qA". Message is assigned a priority of 2000.
 */

RetCode =
    QueSend(QUE_Q_ANY,           /* QSC for single Qid list */
            QueList(qA,
                    QUE_EOL),    /* target Qid */
            "hello world",       /* message text */
            11L,                 /* length of message text */
            2000L,               /* message priority */
            &RetQid,             /* for QueSend return data */
            QUE_WAIT);           /* willing to block */
```

The above operation performs both the text writing and the header placement portions of the message dispatch operation.

As with QuePut(), the `QUE_REPLICATE` option, when specified with QueSend(), 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 only to those processes which are waiting *at the time* of the QueSend() operation.

Example:

```
/*
 * Send a copy of the same message to ALL users
 * waiting for such a message.
 */

RetCode =
    QueSend(QUE_Q_ANY,           /* QSC for single Qid list */
            QueList(qA,
                    QUE_EOL),    /* target Qid */
            "hello world",       /* message text */
            11L,                 /* length of message text */
            2000L,               /* message priority */
            &RetQid,             /* for QueSend return data */
            QUE_REPLICATE);      /* send multicast message */
```

QueSend() is usually used at the beginning of message's existence. Subsequent operations on the message might include a series of QueGet() and QuePut() operations to affect inter-queue routing. The last operation will finally remove the header and its text, using either a QueGet() with a QueRead() or QueReceive() alone.

Example:

```
QueSend(), [ QueGet(), QuePut(), ... QuePut(), ] QueReceive()
```

or

```
QueSend(), [ QueGet(), QuePut(), ... QuePut(), ] QueGet(), QueRead()
```

The rules and use of arguments described above by QueWrite() and QuePut() apply equally to QueSend() where relevant.

The above example demonstrates QueSend() using synchronous blocking. Asynchronous blocking is also possible by specifying one of the three asynchronous blocking options. Refer to the Advanced Topics chapter for a detailed description of the asynchronous blocking options.

2.3.10 QueReceive() - Receiving a Message from a Queue

In a similar vein, it is sometimes not necessary to use QueGet() followed by QueRead() to retrieve a message. A single function, QueReceive(), can be used when the two stages of control provided by QueGet() and QueWrite() are not needed.

"QueGet() + QueRead() = QueReceive()" summarizes the functionality of the QueReceive() function.

Not surprisingly, the pros and cons for using QueReceive() are comparable to those given regarding QueSend().

Advantages of using QueReceive() include:

- No need to manage message headers.
- No need to interact directly with the message text pool.
- Generally slightly better performance than "QueGet() + QueRead()".

Disadvantages include:

- An inability to manipulate message headers independent of their associated texts segments; inter-queue routing using QueReceive() is therefore inefficient.

QueReceive() takes seven arguments:

- A Queue Select Code for choosing one of the candidate messages.
- A QidList identifying candidate messages for each of the listed queues.
- A pointer to a buffer to receive the message text.
- A long integer specifying the maximum size text to copy into the buffer (usually the size of the buffer).
- A pointer to a variable that is assigned the retrieved message's priority.
- A pointer to a Qid variable that gets set by QueReceive() to the actual target Qid chosen. (This poitner can be `NULL` if no return value is desired.)
- A blocking option code in case the operation needs to block.

Example:

```
/*
 * Receive the "hello world" message sent onto queue "qA"
 * in the section describing QueSend. Note: QidList
 * simplification is employed in this example. Since no MSC
 * is applied to 'qA' in the QidList, the HP criteria
 * specified as the QSC is applied implicitly to queue 'qA'
 * as its message select code.
 */

MsgLen =
    QueReceive(QUE_Q_HP,             /* QSC for single Qid list */

               QueList(qA, QUE_EOL),   /* source Qid */

              MsgBuf,     /* message buffer */
                sizeof(MsgBuf),       /* buffer size  */
                &Priority             /* set to message priority */
                &RetQid,          /* returned by QueReceive */
                QUE_WAIT);            /* willing to block */
```

The above operation performs both the header access and the text reading portions of the message retrieve operation.

As was shown in the section describing QueSend(), QueReceive() is most often used at the end of a message's existence. Using QueReceive() with QueSend() to implement queue switching is inefficient due to the wasteful message pool read and write that would occur.

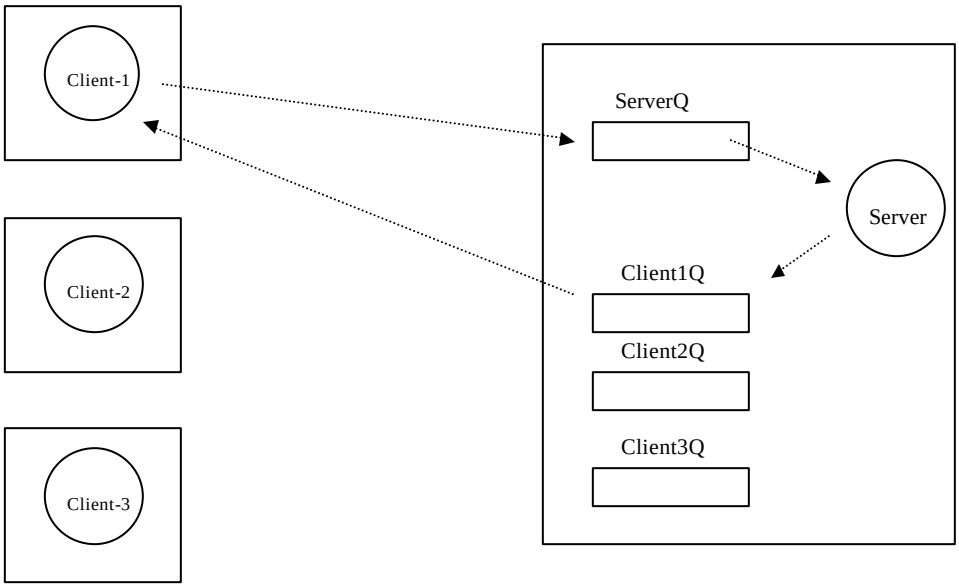
Once again, the rules and use of arguments described by QueGet() and QueRead() apply to QueReceive() where relevant.

The above example demonstrates QueReceive() using synchronous blocking. Asynchronous blocking is also possible by specifying one of the three asynchronous blocking options. Refer to the Advanced Topics chapter for a detailed description of the asynchronous blocking options.

2.3.11 QueSendReceive() - Perform Generic Request/Response

The QueSendReceive() function performs similarly to the RPC request/response paradigm. However, 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.
QueSendReceive() takes the arguments of QueSend() and QueReceive(), each of which is described above.
The usage of queues within QueSendReceive() is highly flexible, supporting Queue Select Codes for the QueSend() and the QueReceive() operations independently. 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).
The following is an example of how the QueSendReceive() function may be used for developing the client side of a client/server application. Consider the following diagram:



There are three clients communicating with a server program. The clients send their requests to the server via the “ServerQ” message queue. Upon sending their request message, the clients await their response on their individual client queue. (The individual client queues may actually be queues that are unnamed, i.e., having the QUE_PRIVATE name.)
The following code segment demonstrates a client’s utilization of the QueSendReceive() call for interacting with the server.

```
/*
 * Client-side code for interacting with server.
 * (Error-handling is not included).
 */

QUE_SEND_ARGS   SendArgs;
QUE_RECV_ARGS   RecvArgs;
XINT    ServerQid, ClientQid;
XINT    RetSendQid, RetRecvQid;
XINT    RetRecvPrio;
struct { . . . } RequestMsgBuffer;
struct { . . . } ResponseMsgBuffer;
```

```
/*
 * Get the Qids to be used for the exchange of
 * messages with the server.
 */

ServerQid = QueAccess("ServerQ");
ClientQid = QueCreate(QUE_PRIVATE, 10, 100);



/*
 * Prepare the arguments for the Send portion of
 * the QueSendReceive() operation.
 */
SendArgs.QueSelectCode = QUE_Q_ANY;
QueListBuild(SendArgs.QidList, ServerQid, QUE_EOL);
SendArgs.MsgBuf = &RequestMsgBuffer;
SendArgs.MsgLength = sizeof(RequestMsgBuffer);
SendArgs.Priority = 100;
SendArgs.QidPtr = &RetSendQid;


/*
 * Prepare the arguments for the Receive portion of
 * the QueSendReceive() operation.
 */

RecvArgs.QueSelectCode = QUE_Q_EA;
QueListBuild(RecvArgs.QidList, ClientQid, QUE_EOL);
RecvArgs.MsgBuf = &ResponseMsgBuffer;
RecvArgs.MsgLength = sizeof(ResponseMsgBuffer);
RecvArgs.Priority = &RetRecvPrio;
RecvArgs.QidPtr = &RetRecvQid;


/*
 * Issue QueSendReceive() call to send request msg onto
 * the ServerQ, and wait for response on the client's queue.
 */

QueSendReceive( &SendArgs, QUE_WAIT, &RecvArgs, QUE_WAIT);
```

Refer to the companion Reference Manual for further detail.

2.3.12 QueCopy() - Copying All or Part of a Message's Text from the Text Pool

QueCopy() copies all or part 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, either the message header on the queue or the copy that was removed via QueGet() with the QUE_NOREMOVE option. Unlike QueRead(), QueCopy() does *not* remove the text from the text pool and, therefore, does not decrement the reference count of the associated text block.

QueCopy() takes four arguments:

- A message header pointer.
- The offset into the message's text where the QueCopy() should commence.
- The number of bytes to copy.
- A pointer to a buffer that is to receive the copied text.

QueCopy() can be used in conjunction with QuePointer() for examining the contents of a message in a manner that is not sensitive to the instance's location.
QueCopy() will fail if the specified offset and length arguments target an area that is beyond the message's actual text space.

Example:

```
/*
 * Get a message header off a queue, then examine the contents of
 * the message. If it is not relevant it can be ungotten using
 * QueUnget. This example allows for the possibility that the
 * instance is NOT local.
 */

MSGHDR MsgHdr;
XINT RetCode;
CHAR *p, Buf[512];

RetCode = QueGet (&MsgHdr, ... );

if (RetCode == 0)
{
    /*
     * Get a pointer to the message's text.
     */

    RetCode = QuePointer (&MsgHdr, &p);

    if (RetCode == QUE_ER_NOTLOCAL)
    {
        QueCopy (&MsgHdr, 0L, MsgHdr.Size, Buf);
        p = Buf;
    }

    /*
     * Examine the text using pointer 'p'.
     */

    ...
}
```

2.3.13 QueUnget() - Ungetting a Message Header

In some situations a programmer may wish to return a retrieved message (header) back to the queue from which it was taken, inserting it into its original position relative to other messages on the queue. Using such a capability a program could "reject" a gotten message after looking at it and return it to its queue as if it had never been taken.
Using QuePut() for this purpose does *not* do the job. That is because QuePut() appends its message to the end of the selected target queue's time strand of messages. In most cases, this is not the message's original chronological position.
QueUnget(), however, is designed for this purpose. It returns a retrieved message header to the queue that it was taken from, placing it by time and priority in the same relative position as it was before it was retrieved.

QueUnget() takes one argument:

- A pointer to a message header that was previously gotten off a queue. (Note that a message header *copy* retrieved by a QueGet() call with the QUE_NOREMOVE option will not accomplish the QueUnget() unless the *original* message header was subsequently removed from the queue.)

Example:

```
/*
 * Get the longest waiting ("earliest arrived") highest priority
 * message from queues represented by QidA, QidB and QidC.
 * If the message text starts with "NOT FOR ME", unget it.
 */

QIDLIST QidList;
MSGHDR MsgHdr;
XINT RetCode, RetQid;
XINT Priority;
CHAR *p, Buf[10];

QidList = QueListBuild(
            QUE_M_HP(QidA),
            QUE_M_HP(QidB),
            QUE_M_HP(QidC),
            QUE_EOL);

RetCode = QueGet(&MsgHdr,
            QUE_Q_EA,
            QidList,
            &Priority,
            &RetQid,
            QUE_WAIT);
if (RetCode == 0)
{
    RetCode = QuePointer (&MsgHdr, &p);

    if (RetCode == QUE_ER_NOTLOCAL)
    {
        QueCopy (&MsgHdr, 0L, 10L, Buf);
        p = Buf;
    }

    if (strncmp(p, "NOT FOR ME", 10) == 0)
        QueUnget(&MsgHdr);
}
```

QueUnget() will succeed in returning the message header *even* if the queue involved is currently full. In such a case, the queue will briefly hold more message data than originally configured for. This exception is necessary to guarantee the success of QueUnget(). Otherwise, a retrieved message may never fit back on its queue, particularly when a busy queue is involved.

Example:

2.3.14 QueBrowse() - Browsing a Message Queue

As was shown earlier, it is possible to use QueGet() to access a copy of a message header without actually removing the header from the queue. A possible application of this feature could be to use the accessed message header copy as a reference point for a subsequent QueBrowse() operation.

QueBrowse() takes two arguments:

- The copy of a message header that has not been dequeued.
- The direction of the browse operation.

QueBrowse() returns with a fully functional message header copy, one position in the specified direction, relative to the message header identified by the message header copy parameter. *Because all headers are fully functional, and thus own a copy of the header's*

text, the calling application must free the header's text via a call to QueRead() at the end of the browsing activity.
The message header copy parameter may have been accessed through a QueGet() operation where QUE_NOREMOVE was specified, or via an earlier QueBrowse() operation.

Example:

```
/*
 * Browse the priority strand of messages on queue Qid.
 * The header of the second highest prio message is removed.
 * Release message text when completed.
 */

/*
 * Stop everything.
 */

QueFreeze();

/*
 * Get copy of highest prio header.
 */

QueGet(&MsgHdr,
        QUE_Q_HP,
        QueList(Qid, QUE_EOL),
        &Priority,
        &RetQid,
        QUE_NOREMOVE | QUE_NOWAIT);

/*
 * Move to next message on priority strand.
 */

QueBrowse(&MsgHdr, QUE_PRIO_NEXT);


/*
 * Dequeue this header.
 */

QueRemove(&MsgHdr);


/*
 * Release message text. (Note: Optimized using QUE_TRUNCATE.)
 */


QueRead(&MsgHdr, Buffer, QUE_TRUNCATE(1) );


/*
 * Restart everything.
 */

QueUnfreeze();
```

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 messages exist in the specified direction.

Example:

```
/*
 * Print the messages on queue Qid, in the order
 * that they arrived.
 */

/*
 * Stop everything.
 */

QueFreeze();

/*
 * Get image of oldest message header.
 */

QueGet ( &MsgHdr,
         QUE_Q_EA,
         QueList(Qid, QUE_EOL),
         &Priority,
         &RetQid,
         QUE_NOREMOVE | QUE_NOWAIT );

do
{
    /*                * Print message's text.
        */

    QueCopy (&MsgHdr, 0L, MsgHdr->Size, Buf);
    printf("Message text = %s\n", Buf);

    /*
     * Move to next message on time strand.
     */

    RC = QueBrowse (&MsgHdr, QUE_TIME_NEXT);

} while (RC != QUE_ER_ENDOFQUEUE);



/*
 * Release message text. (Note: Optimized using QUE_TRUNCATE.)
 */
```

```
QueRead(&MsgHdr, Buffer, QUE_TRUNCATE(1) );


/*
 * Restart everything.
 */

QueUnfreeze();
```

2.3.15 Queue Spooling

Queues are typically created with sufficient capacity for handling expected traffic surges. Unfortunately, actual traffic patterns are not always predictable. A queue may periodically be strained beyond its capacity by brief bursts of heavy message traffic.

2.3.15.1 What is Queue Spooling?

The handling of queue overflow messages is one of the more difficult aspects of a system's design. *X•IPC* provides a dynamic spooling mechanism for its queues. Spooling for a queue can be activated or deactivated via program control whenever necessary. Messages attempting to enter a queue that is full and currently spooling are temporarily placed on the queue's overflow spool. The queue is, in effect, given a virtual capacity extension beyond its original size limits. This elasticity guarantees that no messages are lost during peak operating periods.

Programs executing QuePut() operations on a queue that is spooling are guaranteed not to block. If the message queue is full, then dispatched messages are redirected to the queue's spool.

Spooled messages are automatically absorbed into the actual queue as space on the queue becomes available. The whole process is completely transparent to the programs dispatching and retrieving the messages to and from the queue.

2.3.15.2 QueSpool() - Starting Spooling for a Queue

Spooling is controlled on a queue-by-queue basis. QueSys queues are created with their spooling initially 'off.' The QueSpool() function is used for activating and deactivating a queue's spooling.

QueSys spooling is implemented using a series of files. We will see shortly how this actually works. For now it is convenient to associate a single spool file name with a queue's spool, and to assume for the sake of simplicity that all spooling occurring on behalf of a queue occurs in its designated spool file.

QueSpool() takes two arguments:

- The Qid for which spooling is to be activated or deactivated.
- A spool file name when activating; or the `QUE_SPOOL_OFF` code for deactivating.

Example:

```
/*
 * Create a transaction queue with a capacity for 100 messages
 * and 32 K-Bytes. Activate its spooling so that overflow messages
 * during heavy traffic surges are not lost.
 */

Qid = QueCreate("TransQueue", 100, 32768L);

RetCode = QueSpool(Qid, "/tmp/tqspool");
```

Spooling does not change a queue's internal capacity. The "TransQueue" created above continues to have room for 100 messages and 32 K-Bytes.
Spooling takes place in the file system. As such, starting spooling for a very busy queue will not cause a sudden rush on QueSys resources, such as message text spool space or available message headers. Other programs using other queues in the same QueSys instance are not affected.
When spooling occurs within a network instance it uses the file system of the machine upon which the instance was started. The spool file name argument must therefore conform to the file naming conventions of that platform.
Overflow "TransQueue" messages are automatically placed on the queue's spool file. The spool file can be located anywhere within an accessible file system. The only critical requirement is that the location (i.e., its directory) be read/write enabled for all processes using the queue.

2.3.15.3 QueSpool() - Stopping Spooling for a Queue

Once activated, a queue's spool can grow freely, being bound only by the underlying file system, the operating system or hardware limitations. Consuming messages from a queue will cause spooled messages to be absorbed onto the queue proper from which they too are eventually consumed.
A queue's spooling activity can be deactivated when spooling of overflow messages is no longer desired. Messages already on the spool are unaffected by the deactivation. They continue to be absorbed as space becomes available on the queue. Deactivation of spooling does, however, block any further messages from being written to the queue's spool. QuePut() and QueSend() operations will now block (if so specified) when attempting to dispatch messages to the full queue.

Example:

```
/*
 * Deactivate spooling for the "TransQueue"
 * used in the previous example.
 */

RetCode = QueSpool(Qid, QUE_SPOOL_OFF);
```

Here, the macro QUE_SPOOL_OFF is given as the function's second argument, instead of as a file name. This instructs QueSpool() to discontinue spooling for the given queue. Queue spooling can be started and stopped by an application as often as necessary.

2.3.15.4 The Spooling Mechanism

Knowledge of how QueSys spooling is implemented is not a prerequisite for using it; it is, however, useful to understand in general terms what is happening.
A QueSys spool is maintained using a series of files. The names of these files are based on the spool file name argument given to QueSpool() when spooling for a queue is started. The base name is appended with ".nnn" suffixes, where nnn is an integer between 000 and 999.

Example:

```
/*
 * The following QueSpool call activates spooling for queue 'Qid'.
 * Spooling will be implemented using files:
 * /tmp/spool.000
 * /tmp/spool.001
 * /tmp/spool.002
 * ...
 * /tmp/spool.999
 */

RetCode = QueSpool(Qid, "/tmp/spool");
```

The specified file name is used as a base name for up to one thousand "spool tick files" that will actually hold the spooled messages.
The maximum size of "spool tick files" within an instance is specified within the instance's configuration file using the `SIZE_SPLTICK` parameter. As we saw earlier, `SIZE_SPLTICK` is specified in units of K-Bytes. No spool tick file will grow beyond `SIZE_SPLTICK` bytes in size within an instance.
Spool tick files are maintained as long as they contain live messages. Once all of their messages have been absorbed by the queue, the files are deleted.

Example:

Consider the following situation. A queue has spooled some of its messages. The front of the live messages is in spool tick file 002. They extend into spool tick file 005.

A number of points can be made:

- The size of each tick file does not exceed the SIZE_SPLTICK parameter setting in the configuration file.
- Messages are placed on the queue at the REAR spool tick file. If the REAR file has hit SIZE_SPLTICK bytes in size, the next tick file in the sequence (i.e., 006) is created and it starts accepting spooled messages.
- Message absorption into the queue is done from the FRONT spool tick file. When the current file (i.e., 002) is exhausted of live messages, it is deleted and the next file in the sequence (i.e., 003) becomes FRONT.
- Too large a value for SIZE_SPLTICK will result in wasted file space, holding a queue's old spooled data. For example, the messages at the beginning of file 002 are dead.
- Too small a value for SIZE_SPLTICK will generally cause more spool tick files to be created for each queue.

Spooled messages are never split across two adjacent spool tick files. As a result, SIZE_SPLTICK must at least as large as the largest message to be spooled. *In fact, it should exceed the largest message by at least 32 bytes.*

For obvious reasons, spool tick files should not be removed while spooling is in progress. It is also a hazardous practice to have different queues share the same spool file name. Spooling, when used properly, can add significant flexibility and robustness to message-intensive systems.

2.3.16 QuePurge() - Purging a Queue

Occasionally it is necessary to purge a queue of all its messages. This is accomplished using the QuePurge() function.

QuePurge() deletes all messages held on a specified queue. It also destroys any spooled messages associated with the queue.

In purging the queue, two steps take place:

- All messages on the queue (internal and spooled) are purged.
- All blocked QueGet(), QuePut(), QueSend() and QueReceive() operations involving the purged queue are cancelled and returned with RetCode = QUE_ER_PURGED. "RetQid" is set with the Qid of the purged queue.

The best way to understand QuePurge() is to view it from its aftermath. The resulting queue is left in a state very similar to that of a newly created queue. The only differences is that QuePurge() does not turn spooling off (if it is on), and that the queue's traffic history statistics are left intact. In every other way the resulting queue is like a brand new queue.

QuePurge() takes one argument:

- The Qid of the queue to be purged.

governed by the privileges of all the section involved. More specifically, a user attempting to access a byte of segment data that has multiple sections defined over it, must have the appropriate access privilege through *all* of the sections overlaying that byte.

The memory segment currently has six distinct areas of accessibility, based on the way the three sections overlay the segment and overlap each other. The following matrix summarizes the current read-write accessibility status of the instance's users vis-à-vis the six marked areas of segment data:

	[1]	[2]	[3]	[4]	[5]	[6]
User 3	RO	RO	RO	RW	RO	NA
User 5	RO	RO	**NA**	NA	RW	NA
User 14	RW	RO	NA	NA	RO	NA
Others	RO	RO	NA	NA	RO	NA

As an example, examine user 5's access to the area of segment data identified as [3]. Since area [3] of the segment has two sections defined over it (sections B and C), user 5's access is determined by the privilege settings of both these sections. User 5 is the *owner* of Section C, thus his access to the underlying segment data through Section C is RW. His access through Section B is as an *other*, and is thus NA. Because area [3] is the overlap of sections B and C, user 5's read-write accessibility to the area must satisfy both of these sections' current privilege settings; this is accomplished by satisfying the more stringent of the two. User 5's read-write accessibility to the [3] area is thus NA.

This example was contrived to demonstrate a large range of functional possibilities in a single situation. In fact, many of the access control and synchronization concepts demonstrated can be applied to common application situations.

Example:

Consider a Customer Table that is to be maintained in an X•IPC shared memory segment. Let us assume that the appropriately sized MemSys memory segment (Segment 1) has been created by an initialization program. MemSys user 4 is now ready to initialize it with the most recent Customer Table data, perhaps saved on disk from the end of the previous day's business.

User 4 first defines a section over the entire Customer Table segment and sets its privileges RW for him (as *owner*) and NA to *other*s. This gives user 4 the ability to initialize the table while preventing other users from accessing it in the middle of the initialization process. We will shortly see that this form of section "locking" can be effected with a single MemSys operation.

Once the initialization of the Customer Table is complete, user 4 sets the *other* privileges of the section to RW. This makes the table read-write accessible to other users so that they can make updates as the day progresses. User 4, the initialization program, can now relinquish ownership of the original section.

Now, assume the following update scenario. User 2 wants to examine customer records 100 through 200 as a single unit, while protecting them temporarily from other user write access. To do this, user 2 defines a new section over records 100 through 200, and becomes the section's owner. He then sets the section *owner* privileges to RW and *other* privileges to RO. User 2 can now examine records 100 through 200 with the assurance that they will not change before his eyes. Other users can continue to read the focused records and read-write the remaining records in the table.

Now suppose that user 2, as a result of his examination, decides to update record number 187. He would lock the record by defining another section over that record alone, and setting its privileges appropriately.

At this point user 2 has exclusive access to record 187. Other users continue to have read-only access to the range of records (100-200) being examined by user 2, and read-write access to the remainder of the Customer Table.

MemSys operations that are prevented from succeeding because of impeding section privileges on the segment can opt to block and wait until section privileges exist that permit their operation to succeed.

Thus, had MemSys user 15 initiated a MemRead() operation that was to read Customer record 187, it would block until that area of the Customer Table segment became readable by him. This would occur in one of two ways: User 2 removes the sections he had defined over the Customer Table as described. With only the original section remaining, user 15 would then be permitted to complete his MemRead() operation since its target data (record 187) would be deemed once again as read-accessible by him. Alternatively, user 2 can otherwise modify the privileges settings of his sections, to make them readable by *other*s. This too would unblock user 15.

3.1.4 Segment Data 'Locking' and 'Unlocking'

The most typical form of segment access control activity is for the purpose of gaining exclusive access to a portion of a segment's data for a brief period of time, perhaps to work with that part of the segment in an atomic manner. This type of activity is referred to as "locking."

A pair of functions, MemLock() and MemUnlock(), are provided by MemSys to support the locking and unlocking of segment areas. These two functions are actually implemented using sections, ownership and access privileges as described above. As such they add nothing fundamentally new to the basic MemSys functionality. MemLock() and MemUnlock() are just convenient shorthand function calls that actually call the basic MemSys functions needed to achieve the locking effect.

A segment area becomes "locked" by a user in three steps:

- A section is defined over the targeted area (if one does not exist yet).

- The user becomes owner of the section.
- The section's access privileges are changed to `MEM_RW` for *owner* and `MEM_NA` for *other*.

This is precisely what MemLock() does when "locking" segment data. First, it defines a section over the targeted area (if no such section exists). Then it attains ownership of the section, blocking if necessary. Finally, it sets the section's privileges to RW for the calling user and NA for all others. The net effect is that the specified part of the segment's data is now locked for the caller's exclusive use.
MemUnlock() accomplishes its work by undoing the steps taken by MemLock().

3.1.5 Atomic Read and Write Operations

MemSys read and write operations are guaranteed to be atomic in nature. User programs having read or write access to a targeted area of a segment are guaranteed that their read/write operations will execute atomically, regardless of the amount of data involved. X•IPC MemSys enforces all serialization of segment access when necessary.

3.1.6 Operation Blocking

Complementing the synchronization mechanisms described above, MemSys provides optional blocking for individual read, write or lock operations. A caller program that is currently unable to complete one of these operations due to an access barrier imposed by one or more sections can opt to block until conditions permit the operation to complete. In this way, programs can be automatically synchronized when competing to read, write or lock overlapping areas of a MemSys segment. Synchronization is enforced entirely by MemSys.

3.1.7 Memory Pool

MemSys segments, when created, are allocated from the instance's memory pool , and are returned to the memory pool when deleted.
Two observations follow. First, no MemSys segment can be created larger than the configured size of the memory pool. And second, the aggregate of all MemSys segments used simultaneously by an instance cannot exceed the size of the pool.
The size of an instance's memory pool is specified within the instance's configuration file.

There are two aspects to memory pool configuration:

- The size of the pool.
- The allocation unit used by the pool.

3.1.7.1 Sizing

The memory pool size defines the total amount of memory allocated to the instance for creating MemSys segments. The given value should be reasonably close to the actual shared memory requirements of the instance.
A simple rule for estimating an efficient memory pool size value is given in the discussion on MemSys configuration below.

3.1.7.2 Allocation Unit Size

The second component of memory pool configuration is the size of the pool's allocation unit (i.e., its *tick size*). This value specifies the multiple by which all memory allocations to created segments are made.
An instance working exclusively with small MemSys segments should configure this parameter to a similarly small value. An instance working with large segments can configure this parameter to a large value, although a small value will usually work as well. An instance working with a wide range of segment sizes should opt for a parameter value close to the small end of the range.
A simple formula for choosing a MemSys allocation unit size is provided below.

3.2 MemSys Configuration

The MemSys section of an *X•IPC* instance configuration file describes the composition and capacity of the instance's MemSys.
Six parameters must be set within the MemSys section of the instance configuration file. Additional operating system specific parameters (if required) are described in the relevant Platform Notes.

The configuration parameters are:

- **MAX_USERS**, The maximum number of concurrent users. 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.
- **MAX_SEGMENTS**, The maximum number of concurrent segments. Should be set based on the requirements of the programs using the instance.
- **MAX_NODES**, The number of nodes. MemSys nodes are used internally for tracking users that block on MemSys operations. As with SemSys and QueSys, there is no firm rule for calculating a value for **MAX_NODES**; it depends largely on the nature of the programs that will use the instance. A conservative estimate to start with is:

  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.

- **MAX_SECTIONS**, The maximum expected number of sections that will exist concurrently in the instance. A starting formula for **MAX_SECTIONS** is:

  MAX_SECTIONS = (**MAX_SEGMENTS** * *AverageSegmentSections*)

where:

AverageSegmentSections is as defined above.

- **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.

  SIZE_MEMPOOL 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 comes to 1948 bytes, then 2 K-Bytes should be specified).
- **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.

Example:
Consider the configuration for an instance's MemSys that will support an image processing server application.

Assumptions:

1. There will be between 5 and 10 users and/or MemSys asynchronous operations within the instance at any one time.
2. There will be no more than 8 segments active at any one time.
3. The average number of sections per segment is 25.
4. The expected average segment size is 50,000 bytes.
5. It is estimated that most segments will range in size between 12,800 and 102,400 bytes, with 25% of them being less than 20,000 bytes in size. A safe *25PercentileSegmentSize* value is then 20,000.

Then:

MAX_USERS can be safely set at 10. Little space is required for configuring extra users, so it pays to play it safe.

MAX_SEGMENTS can be set at 8. The MAX_USER reasoning is valid here too.

MAX_NODES follows then as: (8 * 10 * 25) + (10 * 4) + 8 = 2,048.

MAX_SECTIONS calculates as: (8 * 25) = 200.

SIZE_MEMPOOL would be calculated as: (8 * 50,000) = 400,000. This should be rendered as 400K bytes.

SIZE_MEMTICK would be set to 20,000 bytes.

```
#============================================================
#
# File: /projects/local/image.cfg
# Created: May 31, 2001
#
#------------------------------------------------------------
#
# This XIPC instance supports a high-performance
# transaction processing application.
# Note: The instance is defined so that it only
# supports XIPC shared memory. SemSys and QueSys
# (and MomSys) are defined as NULL, by virtue of
# not being included.
#
#------------------------------------------------------------

[MEMSYS]
MAX_USERS   10
MAX_SEGMENTS    8
MAX_NODES   2048
MAX_SECTIONS    200
SIZE_MEMPOOL    400     */ in kb */
SIZE_MEMTICK    20000
#============================================================
```

A further note about MemSys configuration: the above formulae and rules generally produce acceptable parameter values. The values should however be adjusted as necessary based on empirical observations using the MemSys monitor.

3.3 MemSys Functions

3.3.1 MemCreate() - Creating a New Segment

The first step in using a MemSys segment within an instance is to create the segment. MemCreate() takes two arguments:

- ☐ The name of the new segment.
- ☐ A value specifying the size (in bytes) of the segment.

MemCreate() returns the "MemSys segment id" (Mid) of the newly created segment. This value is used as the segment's "handle" in all subsequent MemSys function calls that refer to this segment.

Example:

```
Mid = MemCreate("CustomerTable", 1024L);
```

In the above example, the calling user attempts to create a new segment having the name `"CustomerTable"`. The new segment will be 1024 bytes in size.
A more likely scenario: There is a typedef that defines a customer record type (e.g., `CUSTOMER_RECORD`) and a customer table is being allocated to support a certain number of customers (e.g., `100`).

Example:

```
/*
 * Create a MemSys shared memory segment that will serve
 * as a Customer Table having capacity for 100 customers.
 */

typedef struct
{
 ...
 ...
} CUSTOMER_RECORD;

XINT RecSize = sizeof(CUSTOMER_RECORD);
XINT NumRecs = 100L;

CustTableMid = MemCreate("CustomerTable", NumRecs * RecSize);
```

A note regarding segment creation: duplicate segment names are not allowed within an instance.
Specifying `MEM_PRIVATE` as the name of the new segment creates a segment inaccessible via MemAccess(), effectively making its Mid private to the creating program. Of course, the creating program can pass the 'Mid' to others if it so wishes. The advantage of using `MEM_PRIVATE` as a name is that it is guaranteed not to conflict with any segment name currently in the instance.

3.3.2 MemAccess() - Accessing an Existing Segment

Once a segment has been created, other users can access it (i.e., its Mid) using MemAccess().

MemAccess() takes one argument:

- The name of an existing message segment.

MemAccess() returns the "MemSys segment id" (Mid) of the desired segment. This value is used as the segment's "handle" in all subsequent MemSys function calls that refer to this segment.

Example:

```
CustTableMid = MemAccess("CustomerTable");
```

The above example accesses the `Customer Table` segment created in the previous section.

3.3.3 MemWrite() - Writing Data to a Memory Segment

Writing data to a MemSys memory segment is accomplished via the MemWrite() function call. MemWrite() copies the specified data directly into the targeted memory segment area. The write operation is guaranteed to be executed atomically. Synchronization between competing users is handled automatically by MemSys. MemWrite() attempts to write data into a specific area of a memory segment. This area must be write accessible by the calling user. More precisely, every byte of the targeted area must be write accessible by the caller at the time of the MemWrite() call.

MemWrite() takes five arguments:

- The Mid of the memory segment to be written to.

- The offset into the segment where the MemWrite() operation should commence.
- The number of bytes to be written (the target area size).
- A pointer to the data to be written, or a call to the MEM_FILL macro. MEM_FILL when specified, identifies a byte value with which to fill the entire targeted area.
- A blocking option code in case the operation needs to block.

Example:

```
/*
 * Write "Hello World" into the first 11 bytes of the memory
 * segment identified by Mid. Note: The offset and the size
 * are 'XINT's.
 */

RetCode = MemWrite(Mid, 0L, 11L, "Hello World", MEM_WAIT);
```

Example:

```
/*
 * Write NewCustomerRecord into the 25th entry
 * of the Customer Table that we created earlier.
 */

typedef struct
{
 ...
} CUSTOMER_RECORD;

CUSTOMER_RECORD NewCustomerRecord;
XINT RecSize = sizeof(CUSTOMER_RECORD);
...
RetCode = MemWrite(
      CustTableMid,             /* Target memory segment */
      (XINT)(24 * RecSize),     /* Offset past 24 customers */
      (XINT)RecSize,            /* Target area length */
      &NewCustomerRecord,       /* The new 25th customer entry */
      MEM_WAIT);                /* Block if target area busy */
```

The segment area that is designated to receive the written data must be write-accessible by the calling user. If any byte of the targeted area is not write-accessible, the MemWrite() operation will not succeed.
If, in such a case, MEM_WAIT is specified as the function call's blocking option, then the function will block and wait until the entire targeted area can be written to, at which point it will complete.
If MEM_FILL is given as the data buffer argument, then the specified byte value is written to the entire targeted memory area.

Example:

```
/*
 * Create a 4K shared image segment, define a writeable section
 * over it and then initialize the segment's entire contents to
 * Hex "FF". NOTE: MemSecDef() is described later in the guide.
 */

/*
 * Create the MemSys memory segment.
 */

Mid = MemCreate("ImageSegment", 4096L);

/*
 * Define a section over the entire segment making the segment
```

```
     * read-write accessible. (MemSecDef() and MemSection() are
     * described below.)
     */

    RetCode = MemSecDef(MemSection(Mid, 0L, 4096L));


    /*
     * Fill the entire segment with hex FF.
     */

    RetCode = MemWrite(Mid, 0L, 4096L, MEM_FILL(0xFF), MEM_WAIT);
```

An important efficiency consideration regarding MemWrite() is the following:
If the entire targeted area is "locked" by the writer (i.e., all overlaying sections are owned by the writing user and have *other* privileges of MEM_NA), then the atomic nature of the write operation is guaranteed implicitly and the actual data transfer is performed in its most efficient form, without the need for explicit protection by MemSys.
If, however, any part of the targeted write area is not currently "locked" from other user access (i.e., one or more of the overlaying sections are either not owned by the writing user or are owned but do not have *other* privileges set to MEM_NA), then the atomic nature of the write operation is explicitly enforced by MemSys.
Building on the earlier examples, consider the situation where customer records 100 through 200 have to be updated atomically. The MemWrite() operations will be executed without any need for MemSys to provide explicit synchronization. This is because the calling program has attained exclusive rights to the segment area involved (via the MemLock() call).
Had the targeted area of the write operations not been "locked," then MemSys would have provided the necessary explicit synchronization for each MemWrite() operation in order to ensure its occurring atomically. This would have borne the necessary overhead and would have been somewhat less efficient.

Example:

```
typedef struct
{
    ...
    ...
} CUSTOMER_RECORD;

CUSTOMER_RECORD NewCustomerRecord;

XINT RecSize = sizeof(CUSTOMER_RECORD);

    ...

/*
 * 'Lock' records 100 - 200 for updating purposes. (MemLock() will
 * be described below).
 */

RetCode = MemLock(...);

/*
 * Update the locked records.
 */

for (i = 100; i <= 200; i++)
```

```
{
    /*
     * Update the i-th record.
     */

    ...



    RetCode = MemWrite(
        CustTableMid,                   /* Target memory segment */
        (XINT)((i-1) * RecSize),        /* Offset past i-1 records */
        (XINT)RecSize,                  /* Target area size */
        &NewCustomerRecord,             /* The new i-1th customer entry */
        MEM_WAIT);                      /* Block if target area busy */

}
```

The above examples demonstrate MemWrite() using synchronous blocking options. Asynchronous blocking is also possible by specifying one of the three asynchronous blocking options. Refer to the Advanced Topics chapter for a detailed description of the asynchronous blocking options.

3.3.4 MemRead() - Reading Data from a Memory Segment

Reading data from a MemSys memory segment is accomplished via the MemRead() function call. The read operation is guaranteed to be executed atomically. Synchronization between competing users is handled automatically by MemSys. MemRead() attempts to read data from a specific area of a memory segment. This area must be read-accessible by the calling user. More precisely, every byte of the specified area must be read accessible by the caller at the time of the MemRead call.

MemRead() takes five arguments:

- ☐ The Mid of the memory segment to be read from.
- ☐ The offset into the segment where the MemRead() operation should commence.
- ☐ The number of bytes to read (the source area size).
- ☐ A pointer to a data buffer that receives the read data.
- ☐ A blocking option code in case the operation needs to block.

Example:

```
/*
 * Read the "Hello World" message written to memory segment Mid
 * in the previous section into a local buffer. Note: The offset
 * and the size are 'longs'.
 */

CHAR Buffer[11];

RetCode = MemRead(Mid, 0L, 11L, Buffer, MEM_WAIT);
```

Example:

```
/*
 * Read the 25th entry of the Customer Table into the
 * CustomerRecord variable.
 */

typedef struct
{
    ...
} CUSTOMER_RECORD;

CUSTOMER_RECORD CustomerRecord;

XINT RecSize = sizeof(CUSTOMER_RECORD);
...


RetCode = MemRead(
    CustTableMid,           /* Source memory segment */
    (XINT)(24 * RecSize),   /* Offset past 24 customers */
    (XINT)RecSize,          /* Source area size */
    &CustomerRecord,        /* The 25th customer entry */
    MEM_WAIT);              /* Block if record is busy */
```

The segment area that is specified as the source of the read operation must be read-accessible by the calling user. If any byte of the source area is not read-accessible, the MemRead() operation will not succeed.
If, in such a case, `MEM_WAIT` is specified as the function call's blocking option, the function will block and wait until the entire specified area becomes read-accessible by the calling user, at which point it will complete.
The same efficiency considerations described regarding writing "locked" areas of a MemSys segment apply to MemRead() as well. Namely that MemRead() operations on "locked" segment areas are more efficient than similar operations to unlocked areas.
The above examples demonstrate MemRead() using synchronous blocking options. Asynchronous blocking is also possible by specifying one of the three asynchronous blocking options. Refer the Advanced Topics chapter for a detailed description of the asynchronous blocking options.

3.3.5 MemSection(), MemSectionBuild() - Initializing a Section Variable

As was described earlier, many MemSys operations make use of the notion of sections. MemSys sections logically overlay all or part of a MemSys segment.
MemSys provides a 'typedef' defined object called `SECTION` for easy manipulation of sections within a program. MemSection() and MemSectionBuild() are functions that can be used to initialize `SECTION` variables.

MemSection() takes three arguments:

- The Mid of the memory segment to be used.
- The offset into the segment where the section starts.
- The size of the section.

Example:

```
/*
 * Initialize a SECTION variable that will eventually be used for
 * locking the first 1024 bytes of MemSys segment Mid.
 */

SECTION LockSection;

LockSection = MemSection(Mid, 0L, 1024L);
```

In the above example a SECTION variable "LockSection" has been initialized describing a section on the first 1024 bytes of MemSys segment Mid.

It is very important to note that the MemSection() function is not reentrant and should not be used in an environment where reentrant code is required, such as in a threaded environment or in code that handles signals or interrupts. The MemSectionBuild() function should be used in its place.

MemSectionBuild() takes four arguments:

- A pointer to a section variable.
- The Mid of the memory segment to be used.
- The offset into the segment where the section starts.
- The size of the section.

Example:

```
/*
 * Initialize a SECTION variable that will eventually be used for
 * locking the first 1024 bytes of MemSys segment Mid.
 */

SECTION LockSection;

MemSectionBuild(&LockSection, Mid, 0L, 1024L);
```

In the above example, a SECTION variable "LockSection" has been initialized describing a section on the first 1024 bytes of MemSys segment Mid.
Note that, unlike MemSection(), MemSectionBuild() requires that a SECTION variable be defined. MemSectionBuild returns a pointer to the SECTION variable and, like MemSection, it can be used anywhere a SECTION variable is required.
MemSection() and MemSectionBuild() do *not* define a new section to MemSys. They are simply a short-cut for initializing a SECTION variable. A number of MemSys functions expect SECTION variables as arguments. Using MemSection() and MemSectionBuild() on the fly, when invoking these functions, makes working with them easier. We will see examples of this below.

3.3.6 MemListXxx() – Functions for Manipulating Section Lists

MemSys operations that manipulate memory sections do so using memory section lists. Manipulating a single memory section within these functions is accomplished using a single element list. A list of memory sections is referred to as a MIDLIST. A MIDLIST data type is defined for creating and working with MidLists. Functions expecting a list of memory sections as one of their arguments take a MIDLIST data type for this purpose.

There are two functions for building MidLists: MemList() and MemListBuild(). MemList() takes a list of memory sections as its arguments with `MEM_EOL` marking the end of the list. MemList() creates a `MIDLIST` in its internal static area. For this reason, the returned `MIDLIST` can be safely used only once.
MemLock() expects a `MIDLIST` as its second argument (MemLock is described below). In the next example, MemList() is used "on the fly" to create the `MIDLIST` argument for MemLock().
MemUnlock(), like MemLock(), expects a `MIDLIST` as one of its arguments. The `MIDLIST` built for MemLock in the next example must be rebuilt for MemUnlock().
Example:

```
/*
 * Lock memory sections 'a', 'b', 'c' and 'd'. The user can then
 * work with them under his exclusive control. Unlock when done.
 */

SECTION a, b, c, d;

a = MemSection(...);
b = MemSection(...);
c = MemSection(...);
d = MemSection(...);

RetCode = MemLock(MEM_ALL, MemList(a, b, c, d, MEM_EOL), ...);
...

/* Work with the locked sections */
...

RetCode = MemUnlock(MemList(a, b, c, d, MEM_EOL), ...);
```

MemListBuild() takes a `MIDLIST` variable as its first argument. The remaining arguments are a list of memory section variables as described for MemList(). MemListBuild() creates a `MIDLIST` in the user-provided `MIDLIST` variable. This `MIDLIST` can safely be reused by the programmer.

Example:

```
SECTION a, b, c, d;
MIDLIST MidList;

a = MemSection(...);
b = MemSection(...);
c = MemSection(...);
d = MemSection(...);

MemListBuild(MidList, a, b, c, d, MEM_EOL);

/*
 * Lock memory sections 'a', 'b', 'c' and 'd'.
 * The user can then work with them under his exclusive control.
 * Unlock the same sections when done.
 */

RetCode = MemLock(MEM_ALL, MidList, ...);
...

/* Work with the locked sections */
...

RetCode = MemUnlock(MidList, ...);
```

Unlike the previous example, the `MIDLIST` built for MemLock() can be reused by MemUnlock(). In this way the `MIDLIST` needs to be built only once.
A MidList must not exceed `MEM_LEN_MIDLIST` elements. This is usually not a great concern since `MEM_LEN_MIDLIST` is currently defined to be 8.
Two additonal functions, MemListAdd() and MemListRemove(), allow for updating MidLists dynamically, and another function, MemListCount(), allows determination of the number of elements in a MidList..
MemListAdd() is provided to allow the programmer to add sections to an existing MidList (i.e., one that has been created by MemListBuild()). This is a common requirement in situations where the needed MidList must be built dynamically, based on certain run-time conditions.
MemListRemove() is provided to allow the programmer to remove sections from an existing MidList when necessary.
The calling sequence for MemListAdd() and for MemListRemove() is identical to that of MemListBuild(). These too expect a user-provided MidList as their first argument. The listed sections are added to or removed from that MidList.
The following example is similar to the ones above, except that only one memory section is locked at a time, leaving the others unlocked.

Example:

```
SECTION a, b, c, d, LockedSection;
MIDLIST MidList;

a = MemSection(...);
b = MemSection(...);
c = MemSection(...);
d = MemSection(...);

MemListBuild(MidList, a, b, c, d, MEM_EOL);

/*
 * Lock memory sections 'a', 'b', 'c' and 'd', one at a time, in
 * whatever order they become available. The user can then work with
 * each one under his exclusive control, leaving the others free.
 * Unlock each section when done working with it.
 */

while (MemListCount(MidList)>0)
{
    RetCode = MemLock(MEM_ANY, MidList, &LockedSection, MEM_WAIT);
    ...
    /* Work with whichever section was locked (LockedSection),
     * then unlock it and remove it from the MidList
     */
    ...
    RetCode = MemUnlock(MemList(LockedSection, MEM_EOL), NULL);
    MemListRemove(MidList, LockedSection, MEM_EOL);
}
```

3.3.7 MemLock() - Locking Memory Sections

Locking all or part of a MemSys segment can be accomplished using the MemLock() function call. Recall that a section cannot be locked unless and until all of the segment bytes it overlays are read-write accessible by the calling User.

MemLock() takes four arguments:

- A type code indicating the type of lock operation to perform.
- A `MIDLIST` holding a list of memory sections to lock.
- A pointer to a `SECTION` variable that gets assigned by MemLock().
- A blocking option code in case the operation needs to block.

Example:

```
XINT Mid;
SECTION a, b;
SECTION RetSec;
MIDLIST MidList;

/*
 * Create a 4K MemSys segment.
 */

Mid = MemCreate("TestSegment", 4096L);

/*
 * Initialize SECTION variables 'a' and 'b' to overlay the
 * first and last 1K bytes of the created segment
 */

MemSectionBuild(&a, Mid, 0L, 1024L);
MemSectionBuild(&b, Mid, 3072, 1024L);

/*
 * Build a MIDLIST containing memory sections 'a' and 'b'.
 */

MemListBuild(MidList, a, b, MEM_EOL);

/*
 * Lock memory sections 'a' and 'b'. The user can then work
 * with them under his exclusive control. Unlock the same
 * sections when done.
 */

RetCode = MemLock(MEM_ALL, MidList, &RetSec, MEM_WAIT)
...

/*
 * Work with the locked sections
 */
...

RetCode = MemUnlock(MidList, &RetSec);
```

MemLock() attempts to lock a list of memory sections. Section locking can occur in one of three ways:

- **MEM_ANY**: Lock any of the memory sections listed.
- **MEM_ALL:** Lock all of the memory sections listed as they become available (i.e., cumulatively).
- **MEM_ATOMIC:** Lock all of the memory sections listed, waiting until all of them are available at the same time (i.e., atomically).

In the above example, MEM_ALL is specified as the first argument to MemLock(). This instructs MemLock() to lock the listed sections as they become available. Had MEM_ATOMIC been specified then the function would not have locked any of the listed sections until all of the listed sections were accessible for locking at the same time. Of course, had the function specified MEM_ANY, then the function would return as soon as any of the listed sections became lock-able.

There is no significance to the order of section specification within the MIDLIST when employing MEM_ALL or MEM_ATOMIC. Within a MEM_ANY call, the listed Mids are locked in the order listed.
When MemLock() succeeds, "RetSec" is returned identifying the last memory section locked. For single-section operations, this is not very useful information. For MemLock() operations involving multiple memory sections, however, this information can be important. A NULL RetSec argument can be specified.
When MemLock() fails, and the cause of the failure is related to one of the listed memory sections, RetSec is set to identify the problematic section.
The previous example could have been coded in the following manner producing equal results. Note, in particular, how the memory section descriptions are passed to MemList().

Example:

```
XINT  Mid;
SECTION RetSec, TempSec;

/*
 * Create a 4K MemSys segment.
 */

Mid = MemCreate("TestSegment", 4096L);

/*
 * Lock the first and last K bytes of segment Mid. The user
 * can then work with them under his exclusive control.
 * Unlock the same sections when done.
 */

RetCode = MemLock(
    MEM_ALL,
    MemList(
            *MemSectionBuild(&TempSec, Mid, 0L, 1024L),
            *MemSectionBuild(&TempSec, Mid, 3072, 1024L),
            MEM_EOL)),
    &RetSec,
    MEM_WAIT);

...

/*
 * Work with the locked sections.
 */

...

RetCode = MemUnlock(
    MemList(
            *MemSectionBuild(&TempSec, Mid, 0L, 1024L),
            *MemSectionBuild(&TempSec, Mid, 3072, 1024L),
            MEM_EOL)),
    &RetSec);
```

The above examples demonstrate MemLock() using synchronous blocking options. Asynchronous blocking is also possible by specifying one of the three asynchronous blocking options. Refer to the Advanced Topics chapter for a detailed description of the asynchronous blocking options.

3.3.8 MemUnlock() - Unlocking Memory Sections

The inverse of memory section locking is memory section unlocking. As we have already seen in some of the examples, this is accomplished using MemUnlock().
Locked sections of a memory segment must be unlocked in order for underlying data to become once again read-write accessible by other users. Recall that when a section of a memory segment is locked, its *other* privileges are set to `MEM_NA`, meaning that all other users are not able to read, write or lock the segment area involved.
MemUnlock() takes two arguments:

- A `MIDLIST` holding a list of memory sections to unlock.
- A pointer to a variable that gets assigned by MemUnlock().

Example:

```
/*
 * Unlock the first and last 1K sections of MemSys segment
 * Mid. (It is assumed that they were locked earlier on.)
 */

RetCode = MemUnlock(MemList(MemSection(Mid, 0L, 1024L),
                            MemSection(Mid, 3072, 1024L),
                            MEM_EOL),
                    &RetSec);
```

It is, of course, an error to attempt to unlock a memory section not currently locked by the user. In such a case, RetSec would be returned with the identity of the invalid section. It is acceptable to specify a `NULL` RetSec argument.

3.3.9 Memory Section Primitive Functions

MemSys manipulation of memory sections using MemLock() and MemUnlock() is actually achieved via a group of memory section primitive functions. These functions provide the greatest level of control over access to MemSys segment data.

To best understand the relationship between MemLock(), MemUnlock() and the memory section primitive functions, consider the following:

```
MemLock()   =  MemSecDef()  + MemSecOwn() + MemSecPriv()

MemUnlock() =  MemSecPriv() + MemSecRel() + MemSecUndef()
```

MemLock() is conceptually implemented in three steps. First it calls MemSecDef() to define the specified section. The section upon creation has RW-RW privileges and has no owner. MemSecOwn() is then called for acquiring ownership of the section. MemSecPriv() is finally called to change to privileges to RW-NA. The net result is that the specified section of shared memory is now locked by the calling user. MemUnlock(), when called, reverses the process.

3.3.9.1 MemSecDef() - Defining A Memory Section

The MemSecDef() function is used for defining a new section over a specific area of a MemSys memory segment. Defining a section is the first step in gaining access to, or control of, an area of a memory segment.

MemSecDef() takes one argument:

- A `SECTION` variable describing the new section to be defined.

Example:

```
/*
 * Create a MemSys memory segment of size 10K bytes. The
 * segment will be used to hold a table of codes having
 * varying security requirements.
 */

SECTION TopSecret;
XINT Mid, RetCode;
XINT KByte = 1024L;

/*
 * Create the "CodeTable" MemSys segment.
 */

Mid = MemCreate("CodeTable", 10 * KByte);

/*
 * Initialize the "TopSecret" SECTION variable. The
 * "TopSecret" section is to start at the top of the
 * table and be 2K in size.
 */

TopSecret = MemSection(Mid, 0L, 2 * KByte);

/*
 * Define to MemSys the TopSecret section using the
 * TopSecret SECTION variable we just initialized.
 */

RetCode = MemSecDef(TopSecret);
```

A 10K byte MemSys segment named "CodeTable" is created in the above example. A section is then defined over the first 2K bytes of the segment. (We will build on this example in the next few sections.) Memory sections can be owned. A newly defined section initially has no owner. Initial privilege settings of a new section are `MEM_RW` for *owner* and *other*s.

The above example might have been coded more concisely as:

Example:

```
/*
 * Create a MemSys memory segment of size 10K bytes. Then,
 * define a section over the first 2K bytes.
 */

Mid = MemCreate("CodeTable", 10 * KByte);

RetCode = MemSecDef(MemSection(Mid, 0L, 2 * KByte));
```

Recall that all read and write operations to and from a MemSys segment only succeed when the calling users have the proper access to the underlying segment area involved. This implies that no read or write operation can succeed on a segment unless at least one section has been defined over at least some part of the segment.

MemSys segments, when created, have no sections defined over them and are hence initially inaccessible. It is therefore quite common to define a section over all or part of a segment soon after it is created so as to give it some degree of accessibility.
On the other hand, MemLock() automatically defines the sections it intends to lock (if they do not yet exist), as part of its locking function. Section definition is thus not required when reading or writing to and from locked segment areas.

Returning to an earlier example:

```
/*
 * Create a 4K shared image segment, define a section over
 * it and then initialize the segment's entire contents to
 * Hex "FF". Note: New sections have privileges of MEM_RW
 * for Owner and Others.
 */

Mid = MemCreate("ImageSegment", 4 * KByte);

RetCode = MemSecDef(MemSection(Mid, 0L, 4 * KByte));

RetCode = MemWrite(Mid, 0L, 4 * KByte, MEM_FILL(0xFF), MEM_WAIT);
```

Locking a section of a MemSys segment via MemLock implicitly causes the definition of the specified section. The last example could thus have been coded as:

```
/*
 * Create a 4K shared image segment, lock it for exclusive use,
 * initialize the segment's entire contents to Hex "FF" and then
 * unlock it. Note: MemUnlock() unlocks and undefines the section
 * it is passed. (Using MemSecUndef() for manually undefining a
 * section is described below.)
 */
Mid = MemCreate("ImageSegment", 4 * KByte);

RetCode = MemLock(
    MEM_ALL,
    MemList(MemSection(Mid, 0L, 4 * KByte), MEM_EOL),
    &RetSec,
    MEM_WAIT);

RetCode = MemWrite(Mid, 0L, 4 * KByte, MEM_FILL(0xFF), MEM_WAIT);

RetCode = MemUnlock(
    MemList(MemSection (Mid, 0L, 4 * KByte), MEM_EOL ),
    &RetSec);
```

3.3.9.2 MemSecOwn() - Becoming Owner Of Memory Sections

Owning one or more sections of a MemSys segment can be achieved using the MemSecOwn() function call.
As was the case with section locking (i.e., MemLock()), a user cannot attain ownership of a section unless and until all of the bytes overlaid by the section are read and write accessible by the user.

MemSecOwn() takes four arguments:

- A type code indicating the type of MemSecOwn() operation to perform.
- A `MIDLIST` holding a list of memory sections of which to attain ownership.

- A pointer to a `SECTION` variable that gets assigned by MemSecOwn().
- A blocking option code in case the operation needs to block.

Returning to our `CodeTable` example, we now acquire ownership of the `TopSecret` section of the table.

Example:

```
/*
 * Create a MemSys memory segment of size 10K bytes. Then,
 * define a section over the first 2K bytes.
 */

SECTION TopSecret;

Mid = MemCreate("CodeTable", 10 * KByte);

TopSecret = MemSection(Mid, 0L, 2 * KByte);

RetCode = MemSecDef(TopSecret);

/*
 * Now, become owner of the TopSecret section of the
 * CodeTable, (i.e., the table's first 2K bytes.) The
 * user can then modify the privileges of the section
 * as he sees fit.
 */

RetCode = MemSecOwn(
    MEM_ALL,
    MemList(TopSecret, MEM_EOL),
    &RetSec,
    MEM_WAIT);
```

Notice the similarity between the MemLock() and MemSecOwn() calling sequences. In fact, they are identical. The rules describing argument specification for MemLock() apply equally to MemSecOwn(). Refer to the MemLock() description for the details.
The above examples demonstrate MemSecOwn() using synchronous blocking options. Asynchronous blocking is also possible by specifying one of the three asynchronous blocking options. Refer to the Advanced Topics chapter for a detailed description of the asynchronous blocking options.

3.3.9.3 MemSecPriv() - Modify Memory Section Privileges

A user who owns a section may want to modify its read-write privilege settings. This can be accomplished using the MemSecPriv() function.

MemSecPriv() takes three arguments:

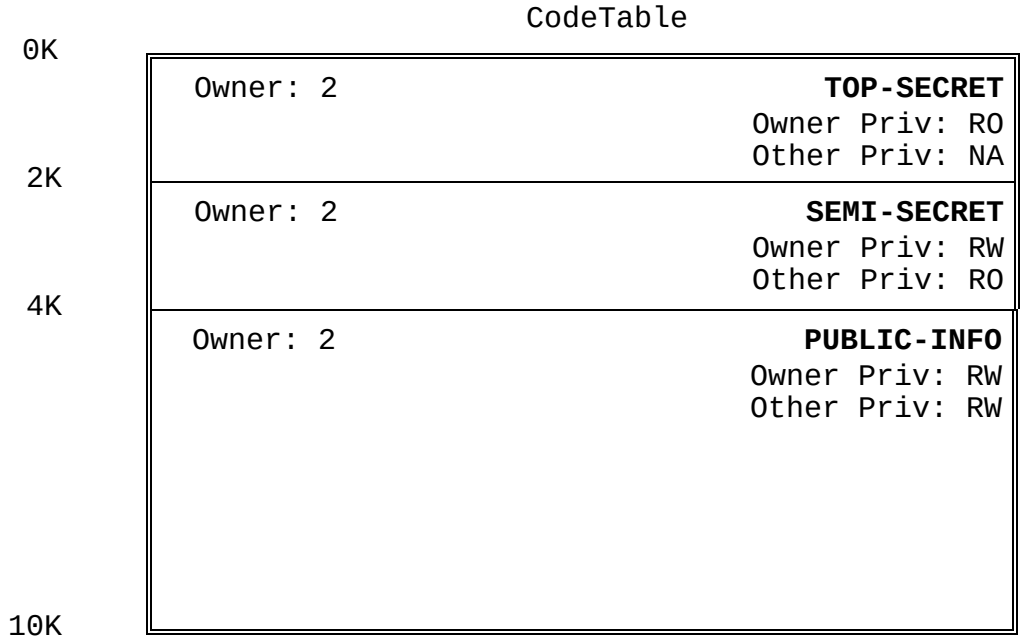
- A `SECTION` variable describing the section whose privileges are to be modified.
- The section's *owner* privilege setting code.
- The section's *other* privilege setting code.

Expanding further on the `CodeTable` examples:

```
/*
 * Create a MemSys memory segment of size 10K bytes. Then,
 *  define a TopSecret section over the first 2K bytes,
 *  define a SemiSecret section over the next 2K bytes,
 *  define a PublicInfo section over the remaining 6K bytes.
 *
 * Attain ownership of the three sections and then set their
 * privileges as follows:
 *  TopSecret: read-only by owner, non-accessible by others.
 *  SemiSecret: read-write by owner, read-only by others.
 *  PublicInfo: should be read-write by all users.
 */

SECTION TopSecret;
SECTION SemiSecret;
SECTION PublicInfo;
XINT Mid;
XINT RetCode;
SECTION RetSec;

Mid = MemCreate("CodeTable", 10 * KByte);

TopSecret  = MemSection(Mid, 0 * KByte, 2 * KByte);
SemiSecret = MemSection(Mid, 2 * KByte, 2 * KByte);
PublicInfo = MemSection(Mid, 4 * KByte, 6 * KByte);

RetCode = MemSecDef(TopSecret);
RetCode = MemSecDef(SemiSecret);
RetCode = MemSecDef(PublicInfo);

/*
 * Become the owner of all three sections of the segment.
 */

RetCode = MemSecOwn(
    MEM_ALL,
    MemList(TopSecret, SemiSecret, PublicInfo, MEM_EOL),
    &RetSec,
    MEM_WAIT);

/*
 * Set the section privileges as specified.
 */

RetCode = MemSecPriv(TopSecret,  MEM_RO, MEM_NA);
RetCode = MemSecPriv(SemiSecret, MEM_RW, MEM_RO);
RetCode = MemSecPriv(PublicInfo, MEM_RW, MEM_RW);
```

Note that the last call to MemSecPriv(), setting the privileges of the PublicInfo section to `MEM_RW` by all, is not necessary since newly defined sections have privilege settings of `MEM_RW` for *owner* and *other* by default.

Assuming that the calling process is MemSys user 2, the resulting situation is as follows:



3.3.9.4 MemSecRel() - Relinquishing Ownership Of Memory Sections

A user can relinquish ownership of one or more sections that he owns, using the MemSecRel() function. The listed sections then become ownerless. Of course, if one or more users are blocked waiting for a chance to own any of the released sections, then the longest waiting user becomes the new owner.

MemSecRel() takes two arguments:

- A `MIDLIST` holding a list of memory sections to relinquish ownership of.
- A pointer to a `SECTION` variable that gets assigned by MemSecRel().

Returning once more to the `CodeTable` example:

```
/*
 * Relinquish ownership of the PublicInfo section.
 */

RetCode = MemSecRel(MemList(PublicInfo, MEM_EOL), &RetSec);
```

3.3.9.5 MemSecUndef() - Undefining A Memory Section

The MemSecUndef() function is used for removing a section definition from a MemSys segment. Undefining a section removes the section and any of its access control influence from the underlying segment.
A user can only undefine a section that he owns or that has no owner. Otherwise, the call will return with an `MEM_ER_ACCESSDENIED` error code.

MemSecUndef() takes one argument:

- A `SECTION` variable describing the section to be undefined.

Example:

```
/*
 * Undefine the "PublicInfo" section.
 */

RetCode = MemSecUndef(PublicInfo);
```

Undefining a section can affect the accessibility to all or part of the MemSys segment area it overlays. For segment areas upon which no other sections are defined, accessibility to that portion of the segment will be impossible until a new section is defined over that area or until the section is implicitly re-defined via MemLock().
For segment areas upon which multiple sections have been overlaid, undefining one of the sections can change the accessibility to the underlying segment data for one or more users, depending on the privilege settings of the section removed and of those that remain.
It is important to undefine a section as soon as it is no longer of use. Extra section definitions over a segment, even if they exert no influence on the accessibility of underlying segment data, can cause some performance degradation.

3.3.10 MemDelete() - Deleting a Segment

A segment should be deleted from its instance when it is no longer needed. This recycles internal MemSys resources and makes the MemView monitor less cluttered.

MemDelete() takes one argument:

- ☐ The Mid of the segment to be deleted.

Example:

```
RetCode = MemDelete(Mid);
```

MemDelete() will succeed only if the subject segment is completely inactive at the time. Segments that have one or more sections defined over it cannot be deleted.
If a segment must be removed regardless of its current status, then MemDestroy() should be employed.

3.3.11 MemDestroy() - Destroying a Segment

A segment that must be removed from its instance can be destroyed using MemDestroy(). MemDestroy() removes the subject segment regardless of the segment's current status.

MemDestroy() takes one argument:

- ☐ The Mid of the segment to be destroyed.

Example:

```
RetCode = MemDestroy(Mid);
```

When a segment is destroyed, a number of things occur:

- ☐ All MemRead(), MemWrite(), MemLock() and MemSecOwn() operations involving the destroyed segment are cancelled and returned with `RetCode = MEM_ER_DESTROYED`.

- All users locking or owning sections over the destroyed segment have those sections removed from their ownership. This occurs silently and it is the responsibility of the program to adjust to the segment's destruction.

For obvious reasons, MemDestroy() should be used sparingly. Its most likely application would be within the execution of a system's "cleanup" program at which time the above side-effects are normally of no concern.

3.3.12 MemInfoSys() - Information About an Instance's MemSys

X•IPC provides a set of MemSys functions that can be used to access status information about various aspects of an instance's MemSys.
The returned data can be used to make run-time decisions about on-going application processing.

MemInfoSys returns with information about the instance's MemSys that the user is logged into. MemInfoSys() takes one argument:

- A pointer to a `MEMINFOSYS` structure that is to be returned filled with the subsystem's status information.

Example:

```
MEMINFOSYS SysData;

RetCode = MemInfoSys(&SysData);
```

A complete description of how to use the Info functions is presented in the Advanced Topics chapter of the X•IPC User Guide, in the section entitled, "Info Function List Manipulation."
The definition of the `MEMINFOSYS` datatype is included in the User Data Structures chapter of the QueSys/MemSys/SemSys Reference Manual.

3.3.13 MemInfoUser() - Information about a MemSys User

MemInfoUser() returns with information about a specified user. MemInfoUser() takes two arguments:

- The Uid whose status is desired.
- A pointer to a `MEMINFOUSER` structure that is to be returned filled with the user's status information.

Besides statistical data, the `MEMINFOUSER` structure returns with "list" data related to the specified user. Each user has an *HList*, *QList* and *WList* associated with it.

- The HList is the list of Sections currently held (owned or locked) by the subject user. The Sections are listed in the order that they were acquired.
- The QList is 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.
- The WList is 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 lists within MEMINFOUSER are arrays that can accommodate up to MEM_LEN_INFOLIST elements. The actual lists may, at times, be greater than MEM_LEN_INFOLIST elements in length. A call to the MemInfoUser() function must therefore be preceded by the setting of three MEMINFOUSER structure members (*HListOffset*, *QListOffset* and *WListOffset*) with values specifying what portions of the three respective lists are desired.
More specifically, before MemInfoUser() is called, the three list offset variables within the MEMINFOUSER structure must be set, indicating from which point in each list to return data. Setting the offsets to zero directs the function to return with list data from the start of the lists.
Example:

```
MEMINFOUSER UserData;

UserData.HListOffset = 0;
UserData.QListOffset = 0;
UserData.WListOffset = 0;

RetCode = MemInfoUser(Uid, &UserData);
```

A complete description of how to use the Info functions is presented in the Advanced Topics chapter of the *X/IPC* User Guide, in the section entitled, "Info Function List Manipulation."
The definition of the MEMINFOUSER datatype is included in the User Data Structures chapter of the QueSys/MemSys/SemSys Reference Manual.

3.3.14 MemInfoMem() - Information about a MemSys Segment

MemInfoMem() returns with information about a specified memory segment. MemInfoMem()_ XE "MemInfoMem()" _ takes two arguments:

- The Mid whose status is desired.
- A pointer to a MEMINFOMEM structure that is to be returned filled with the memory segment's status information.

Besides statistical data, the MEMINFOMEM structure returns with "list" data related to the specified memory segment. Each memory segment has an *SList* and *WList* associated with it.

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

The lists within MEMINFOMEM are arrays that can accommodate up to MEM_LEN_INFOLIST elements. The actual lists may at times be greater than MEM_LEN_INFOLIST elements in length. A call to the MemInfoMem() function must therefore be preceded by the setting of two MEMINFOMEM structure members (*SListOffset* and *WListOffset*) with values specifying which portions of the two respective lists are desired.

More specifically, before MemInfoMem() is called, the two list offset variables within the MEMINFOMEM structure must be set, indicating from what point in each list to return data. Setting the offsets to zero directs the function to return with list data from the start of the lists.

Example:

```
MEMINFOMEM MemData;

MemData.SListOffset = 0;
MemData.WListOffset = 0;

RetCode = MemInfoMem(Mid, &MemData);
```

A complete description of how to use the Info functions is presented in the Advanced Topics chapter of the X•IPC User Guide, in the section entitled, "Info Function List Manipulation."
The definition of the MEMINFOMEM datatype is included in the User Data Structures chapter of the QueSys/MemSys/SemSys Reference Manual.

3.3.15 MemInfoSec() - Information About an Instance's Section

MemInfoSec() returns with information about a Section currently defined in MemSys. MemInfoSec() takes two arguments:

- A SECTION variable identifying the section whose status is desired.
- A pointer to a MEMINFOSEC structure that is to be returned filled with the section's status information.

Example:

```
MEMINFOSEC SecData;

RetCode = MemInfoSec(&SecData);
```

A complete description of how to use the Info functions is presented in the Advanced Topics chapter of the X•IPC User Guide, in the section entitled, "Info Function List Manipulation."
The definition of the MEMINFOSEC datatype is included in the User Data Structures chapter of the QueSys/MemSys/SemSys Reference Manual.

3.3.16 MemPointer() - Accessing a Pointer to a Segment

MemPointer() obtains a pointer to the first byte (offset 0) of the MemSys segment it is passed. The pointer can then be used for directly accessing the data within the segment.

MemPointer() takes two arguments:

- The Mid of the segment whose pointer is desired.
- The address of a pointer variable to be filled with the segment pointer.

MemPointer() returns MEM_ER_NOTLOCAL when the calling program is working within a network instance that was started on another node.
This function is a double-edged sword. On the one hand, it provides the most basic method of manipulating 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 in 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) to access areas of a MemSys segment that are currently "locked" by the user. To use it otherwise could produce unpredictable results at best.

Example:

```
/*
 * Create a Data Segment, lock it and then access a
 * pointer to the segment for manipulating its data.
 * This example assumes that the instance is local.
 */

XINT Mid;
XINT RetCode;
SECTION RetSec;
CHAR *p;

/*
 * Create the MemSys segment.
 */

Mid = MemCreate("Data", 256L);

/*
 * Lock it for exclusive access.
 */

RetCode = MemLock(
    MEM_ALL,
    MemList(MemSection(Mid, 0L, 256L), MEM_EOL ),
    &RetSec,
    MEM_WAIT);

/*
 * Get a pointer to the segment.
 */

RetCode = MemPointer(Mid, &p);

/*
 * Set the bytes of the Data segment to: 0, 1, 2, 3, ...
 * directly, via pointer "p".
 */


for (i = 0; i < 256; i++)
    *(p + i) = (BYTE)i;

/*
 * Unlock it.
 */

RetCode = MemUnlock(MemList(MemSection(Mid, 0L, 256L)), &RetSec);
```

MemPointer() will return a valid pointer to a segment of a 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 the NULL pointer.

A more robust version of the above example would have tested the value returned by MemPointer() to determine whether the instance being used was local or not. Code for handling the non-local case would then be included.

An example follows.

```
/*
 * Create a Data Segment, lock it and then access a
 * pointer to the segment for manipulating its data.
 */

XINT Mid;
XINT RetCode;
SECTION RetSec;
CHAR *p;
CHAR Buffer[256];
XINT RemoteFlag = FALSE;

/*
 * Create the MemSys segment.
 */

Mid = MemCreate("Data", 256L);

/*
 * Lock it for our exclusive access.
 */

RetCode = MemLock(
    MEM_ALL,
    MemList(MemSection(Mid, 0L, 256L)),
    &RetSec,
    MEM_WAIT);

/*
 * Attempt to access a pointer to the MemSys segment.
 */

RetCode = MemPointer(Mid, &p);

if (RetCode == MEM_ER_NOTLOCAL)
{
    MemRead(Mid, 0L, Buffer, 256L, MEM_WAIT);
    p = Buffer;
    RemoteFlag = TRUE;
}

/*
 * Set the bytes of the Data segment to: 0, 1, 2, 3, ...
 * directly, via pointer "p".
 */

for (i = 0; i < 256; i++)
    *(p+i) = (BYTE)i;

/*
 * Update the segment, if it's not local.
 */

if (RemoteFlag == TRUE)
    MemWrite(Mid, 0L, Buffer, 256L, MEM_WAIT);

/*
 * Unlock it.
 */

RetCode = MemUnlock(MemList(MemSection(Mid, 0L, 256L)), &RetSec);
```

3.3.17 MemFreeze() - Freezing MemSys

X•IPC also provides the user with the ability to attain exclusive control over an instance's MemSys. This mechanism allows a user to execute a series of MemSys operations, with the assurance that no other user's MemSys operations are interwoven with his.
Such a capability is important when a user requires exclusive access to the subsystem for a brief period of time. As an example, a need for this feature would arise when writing a function that increments an arbitrary four byte "word" of MemSys shared memory in an "atomic" manner.
One solution would be to lock the targeted bytes using MemLock(), perform the increment operation and then do MemUnlock(). While this would work, an alternative approach would be more efficient in situations where all competing users *always* have read-write access to the targeted area.
In that case, temporarily freezing the subsystem for the duration of the three necessary MemSys operations would work more efficiently.

Example:

```
XINT
MemIncr(Mid, Offset)
XINT Mid;
XINT Offset;
{
    XINT Data;

    MemFreeze();

    MemRead(Mid, Offset, (CHAR *)&Data, 4L, MEM_NOWAIT);
    Data ++;
    MemWrite(Mid, Offset, (CHAR *)&Data, 4L, MEM_NOWAIT);

    MemUnfreeze();

    return (Data);
}
```

A more complete version of this example appears in the Advanced Topics section of this Guide.
A further note regarding MemFreeze(). It is an error for a user to issue a blocking MemSys function call specifying a blocking option code (i.e., `MEM_WAIT` or `MEM_TIMEOUT`) once the user has frozen the subsystem.

3.3.18 MemUnfreeze() - Unfreezing MemSys

MemUnfreeze is the bracketing function to MemFreeze. It returns the MemSys subsystem to its unfrozen state. Other MemSys users resume normal MemSys operations.

Example:

```
MemUnfreeze();
```

MemUnfreeze() will fail if the calling user has not frozen the subsystem.

3.4 The MemSys On-Line Monitor: MemView

MemView is the on-line monitor for X•IPC MemSys.

3.4.1 Starting MemView

MemView is started from the command line using the "MemView" command.

MemView takes two arguments:

- The first argument is the initial "interval" snapshot setting. It defines in milliseconds the initial update frequency of the monitor. The interval argument is mandatory.
- The second argument is the instance file name of the instance to be monitored. This argument is optional. If it is omitted, MemView uses the value of the `xipc` environment variable for the instance file name of the instance to start monitoring.

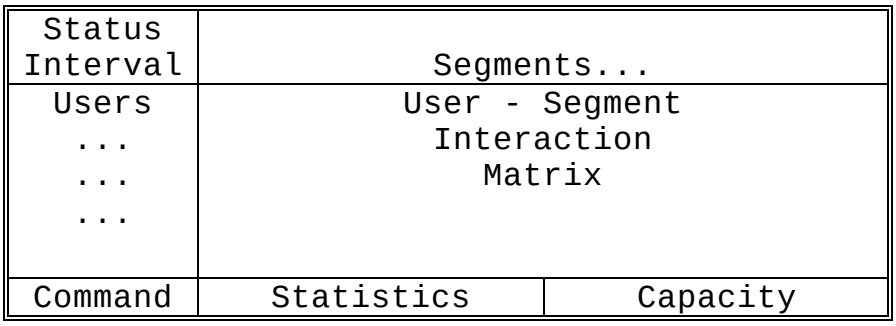
Example:

```
memview 100 /usr/demo
```

The above command starts the MemView monitor for the MemSys subsystem of the `/usr/demo` instance. The initial update interval is set to 100 milliseconds.

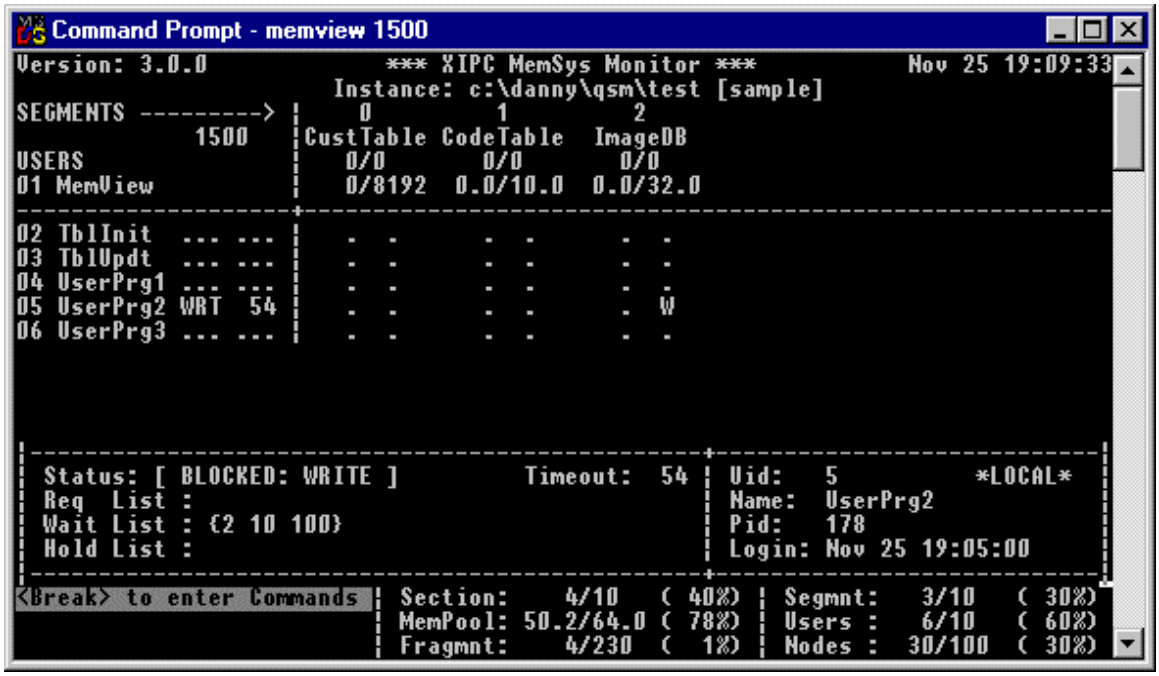
3.4.2 MemView Layout

MemView's main display is matrix-like in appearance. Users logged into the instance and existing MemSys segments form the axes of the matrix. Interaction between instance users and segments is displayed within the body of the "interaction matrix".
MemSys operations that block asynchronously are treated as pseudo-users of MemSys. These *Asynchronous Users* are displayed in the same manner as ordinary users, thus providing a consistent visual display of all pending MemSys asynchronous operations.



Monitor status and interval setting are shown at the top left portion of the screen. MemSys memory pool and other capacity data is displayed at the lower right portion of the screen. The command entry window is at the lower left of the screen.

3.4.2.1 Sample MemView Screen



3.4.2.2 User Entries

Users logged into the instance are listed on the left side of the interaction matrix, one line per user.
Each user entry includes:

- A MemSys user ID.
- The user's login name.
- The user's blocking status (if any).
- The blocking time out value (if any).

An example (not associated with the screen presented above) follows.

```
02 TblInit  ... ...
03 TblUpdt  WRT ...
06 NetProg  RD   27
29 A029-006 ALL  ...
```

In this example, four MemSys users are identified with three ordinary and one asynchronous MemSys operation.

- MemSys user 2 has the login name "TblInit." The user is not blocked on any MemSys operation.
- MemSys user 3 has logged in as "TblUpdt." It is blocked on a MemWrite() operation and is blocked indefinitely, thus having no time out value.

- MemSys user 6, logged in as "NetProg," is blocked on a MemRead() operation and has a time out pending. There are 27 seconds remaining until the operation times out.
- MemSys user 29 is an asynchronous MemLock (MEM_ALL) operation that was initiated by user 6.

3.4.2.3 Memory Segment Entries

The instance's memory segments are identified across the top of the interaction matrix.

Each segment entry includes:

- The Mid of the segment.
- The user-assigned ASCII name of the segment.
- The segment's section status (Locked Sections/Total Sections).
- The segment's byte status (Locked Bytes/Total Bytes).

An example (not associated with the screen presented above) follows.

```
      0              1              5
  CustTable     CodeTable      ImageDB
    4/6             1/1            0/0
  100/8192       1.2/10.0       0.0/64.0
  ...
```

In this example:

- Segment "CustTable" is shown to have a Mid of 0. It is a segment that is 8192 bytes in size. It currently has 6 sections defined over it, of which 4 sections are "locked." The locked sections cover 100 bytes of the segment's 8192 bytes.
- Segment "CodeTable" has Mid 1. Its size is 10K bytes. There is currently one section defined over the segment, it is 1.2K bytes in size and it is locked.
- Segment "ImageDB" has Mid 5. It is being used to hold a memory resident Image Database. The segment is 64K bytes in size. There are currently no sections defined over the segment.

3.4.2.4 Interaction Matrix Cells

Each cell on the MemView interaction matrix describes the current relationship between a user and a segment.

Possible cell values include:

`.. ..`	Indicating that the user is not blocked in any manner on the intersecting segment, nor has he locked any of the segment's sections.
nn mm	Indicating that the user has locked ***nn*** sections on the intersecting segment, and that he is blocked waiting to lock an additional ***mm*** sections on the same segment.
***nn* R**	Indicating that the user has locked ***nn*** sections on the intersecting segment, and that he is blocked waiting to read (R) data to the segment.

nn* W** Indicating that the user has locked ***nn sections on the intersecting segment, and that he is blocked waiting to write (W) data to the segment.

3.4.3 Monitoring Modes

The topic of monitoring modes -- the available options and when they should be used -- is described in detail in the X•IPC User Guide.

3.4.4 MemView Zoom Windows

MemView provides the developer with three zoom window capabilities.

3.4.4.1 Zooming in on a User

The MemView user zoom window creates a detailed display of the status of a particular MemSys user. The command string for user zooming is "zuN" where N is the Uid to be zoomed in on.

Example:
The command for opening a zoom window on the user having a Uid of 4 is:

```
Command> zu4
```

```
Status:  [ NOT BLOCKED ]                  Uid: 4
  Req  List :                             Name: TableDaemon
  Wait List :                             Pid:  23
  Hold List :  (1 100 8)                  Login: Dec 23 12:23
```

User "TableDaemon" has locked one section for its exclusive use. The section is on segment Mid 1, at offset 100 and is 8 bytes long. The user is otherwise not currently blocked.

```
Status:  [ BLOCKED ATOMIC ]               Uid: 4
  Req  List :  (0 0 64)(1 0 64)           Name: TableDaemon
  Wait List :  (0 0 64)(1 0 64)           Pid:  23
  Hold List :  (1 100 8)                  Login: Dec 23 12:23
```

The user has now blocked attempting to lock the first 64 bytes of both segments 0 and 1. The user will not be unblocked until both of the pending sections are available at the same time (i.e., ATOMIC).

```
Status:  [ NOT BLOCKED ]                  Uid: 4
  Req  List :                             Name: TableDaemon
  Wait List :                             Pid:  23
  Hold List :  (1 100 8)(0 0 64)(1 0      Login: Dec 23 12:23
64)
```

The request has been satisfied and the user is no longer blocked.
The three lists included in the MemSys user zoom window serve the same function as their counterparts in the SemSys user zoom window. Refer to the SemSys chapter for an example that fully describes the information being provided.

3.4.4.2 Zooming in on a Segment

The memory segment zoom window provides a complete report of a segment's current status. The command string for zooming on a segment is "zmN" where N is the Mid to be zoomed in on.

Example:

```
The command for opening a zoom window on memory segment 2 is:


   Command> zm2
```

```
Map:                                          Mid: 6
...oooo...o..oo........oooooo....oo.          Name: ImageDB
 Lock:  2000/4096 ( 49%)                      CreateUid: 2
[*******.........]                            Created: Jan 4
 Last Uid:  5     Offset: 1024    Size:       9:30
32
 Wait List :  8  14
```

Memory segment 6 is shown to be 4K bytes in size (4096) and 2000 of the segment's 4096 bytes (49%) are currently locked. The approximate location of the locked sections on the segment are indicated via a map. The 'o' characters mark the relative position and size of locked sections on the segment.
The last user to have locked a section on the segment was user 5. The section locked was at offset 1024 and was 32 bytes in size.
The wait list indicates that users 8 and 14 are currently blocked on operations involving the segment. The details of their individual blockages can be viewed on their respective user zoom windows.

3.4.4.3 Zooming in on Memory Pool Status

A zoom window for monitoring the MemSys memory pool status can be opened using the "zp" command string.

Example:

```
   Command> zp
```

```
Capcty:  21.1/40.0  ( 53%)                    Pool Size: 40K
[*******.......]                              Tick Size: 1024
 Frgmnt:    24/208   ( 12%)
[**............]
 Largst Blk:  29008
```

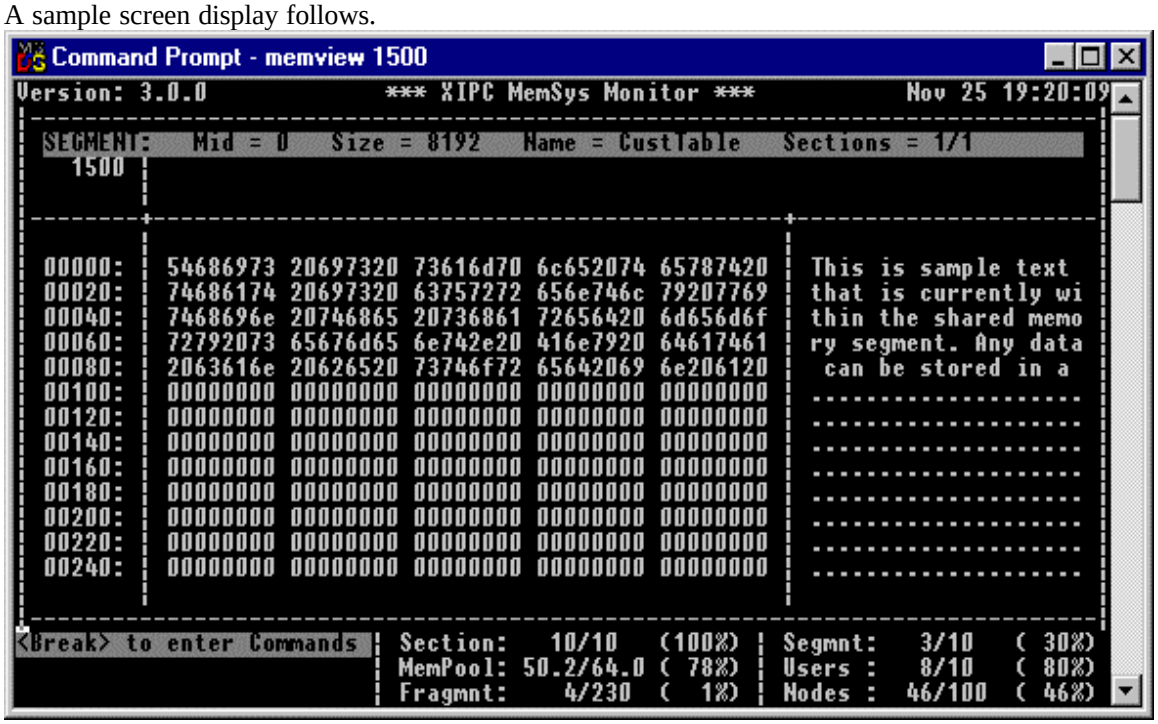
The interpretation of the memory pool window is the same as the QueSys text pool window described in the QueSys chapter.

3.4.5 Watching Memory Segment Contents - The Watch Window

A key element of MemView is the memory Watch Window facility. With it, a developer can view shared memory contents in real-time .
A watch window is opened using the command string "wN," where N is the Mid to be watched.

Unlike the zoom windows already described, the watch window uses the top 3/4 of the monitor screen. The system statistics and command windows remain visible at the bottom of the screen.

A sample screen display follows.



A replication of a different Watch Window is provided below, as an example for reading sample screen data:

Segment 50 Flow *	Mid = 2 Size = 8192 Name = CustTable Sections = 2/4 4W(10 32) 7R(1024 128)

```
 08020:   00000000 00000000 00313030          .........1000011
 08040: 30303131                              Harvey Schmidlap
 08060:   48617276 65792053 63686e69          p 094-40-2819 37
 08080: 646c6170                              Street Lex City.
 08100:   70203039 342d3430 2d323831          07761....23.00..
 08120: 39203337                              .........4311901
 08140:   53747265 6574204c 65782e20          Joe Wilson 072-4
 08160: 43697479                              5-8019 Peteville
 08180:   30373736 31000101 73323347          ............
          30307274
          00000000 00000000 00343331
          31393031
          4a656520 57696c73 6f6e2030
          37322d34
          352d3830 31392050 65746556
          696c6c65
          00000000 00000000 00000000
<Break> to Command    Section:   4/50   ( 8%)    Segmnt: 1/10 (10%)
                      MemPool: 1.1/10.0 (10%)    Users: 11/40 (27%)
                      Fragmnt:   2/36   ( 5%)    Nodes: 37/80 (46%)
```

The watch window can operate in the same update modes as those available from the monitor screen.

The screen example immediately above for example, depicts a watch window monitoring the contents of segment 2, in flow mode, with an interval rate of 50 milliseconds. In such a mode it is possible to watch each and every shared memory update as it occurs, with the updates occurring in slow motion.

The watch window presents the segment's data contents in the same format as used by the QueSys browse facility. The screen is broken into three regions. Offsets appear on the left, segment data in HEX appears in the middle and the same data in ASCII format appears on the right.

The top of the watch window identifies the segment being watched, as well as providing the segment's section statistics.

Also included at the top of the watch window is the list of users (if any) that are currently blocked trying to read or write data to or from the segment being watched.

In this example, user 4 is currently blocked attempting a write operation to the segment. The write operation target starts at offset 10 in the segment and extends for 32 bytes. A second user, user 7, is blocked attempting to read 128 bytes starting at offset 1024 in the segment.

3.4.5.1 Watch Window Commands

MemView commands can be used from within the watch window in the same way that they are used from the main monitor window. Examples:

Command	Effect
`in`	Set the interval to *n* milliseconds
`tf`	Enter trace flow mode
`ts`	Enter trace step mode
`wn`	Open a watch window on segment *n*
`sn`	Open a section window on segment *n*
`bn`	Browse the contents of segment *n*
`q`	Exit the watch window

Additional commands are available that are specific to the watch window. They provide a means for moving the watch window to different parts of the segment. These movement commands are:

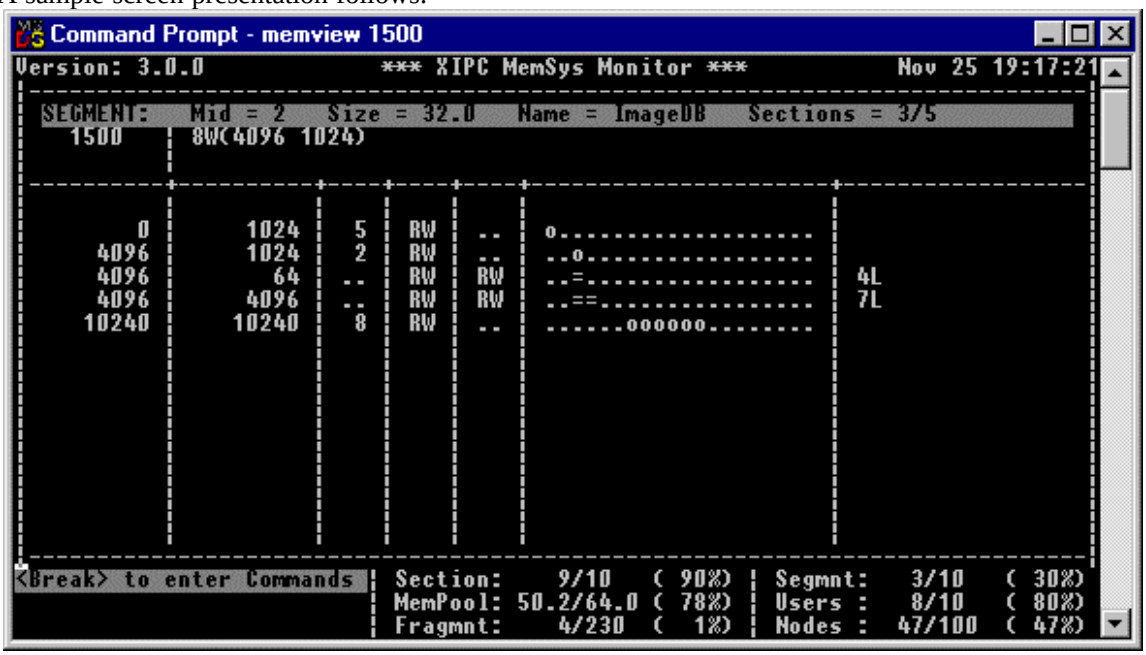
Command	Effect
⇑ (up arrow)	Scroll up one line (20 Bytes)
⇓ (down arrow)	Scroll down one line (20 Bytes)
`PAGE-UP`	Scroll up one page (260 Bytes)
`PAGE-DOWN`	Scroll down one page (260 Bytes)
`HOME`	Scroll to the top of the memory segment
`END`	Scroll to the bottom of the memory segment.

Scrolling only works where it makes sense. Otherwise, the command is ignored.

3.4.6 Monitoring a Segment's Sections - The Section Window

MemView also provides a window for monitoring the details of section activity occurring on a segment This window is the Section Window.
The section window provides a detailed picture of all the sections that are defined on a segment, including a summary of the users that are attempting to Lock or Own any of the segment's sections.
A section window is opened using the command string "sN," where N is the Mid to be monitored.
Like the watch window, the section window uses the top 3/4 of the monitor screen. The system statistics and command windows remain visible at the bottom of the screen.

A sample screen presentation follows.



A replication of a different Section Window is provided below, as an example for reading sample screen data:

Segment 50 Flow *	Mid = 2 Size = 8192 Name = CustTable Sections = 2/4 4W(10 32) 7R(1024 128)					
0 1024 5000 8190	 400 2000 226 2	.. 7 14 0	RW RW RW RW	RW .. .. RO	==..... ooo..... .....oo.... .....=	 5L 1N 8L
<Break> to Command User 14: MemUnlock	Section: 4/50 (8%) MemPool: 1.1/10.0 (10%) Fragmnt: 2/36 (5%)				Segmnt: 1/10 (10%) Users: 11/40 (27%) Nodes: 37/80 (46%)	

The section window also operates in the same update modes as those available from the main monitor window.

The screen example immediately above monitors the section activity occurring on segment 2, in flow mode, with an interval rate of 50 milliseconds. In such a mode it is

possible to observe section related operations as they occur, with updates reported continuously.

The section window presents its information with each row representing a section. For each section, the following information is given:

Offset The offset where the section starts.

Size The size of the section.

Owner The owner of the section, if one exists. ".." indicates a section with no owner.

Owner Priv The section's Owner privilege setting. ".." indicates No Access.

Other Priv The section's Other privilege setting. ".." indicates No Access. A section that is inaccessible by Others is considered locked.

Map A map depicting the size and location of the section relative to the entire segment. Sections that are locked are indicated by a string of 'o' characters. Otherwise the section's relative size and location is marked with '=' characters.

Wait List The list of users currently blocked attempting to either lock or own the section.

In the sample section window, four sections are currently defined over the "CustTable" segment (Mid = 2):

- The first section starts at offset 0 of the segment and is 400 bytes in size. The section is currently ownerless. The privilege setting for both *owner* and *other* is RW. The section, as expected, appears at the beginning of the segment map. Finally, there are no users blocked trying to lock or own the section.
- Next, is a 2000 byte section starting at offset 1024. This section is currently locked by user 7. Thus the privilege settings for *owner* and *other* are RW and NA ("..") respectively. The map indicates the section's location. Users 5 and 1 are currently blocked, trying to lock (L) and own (N) the section respectively.
- The next section is locked by user 14. User 8 is waiting in turn to lock it. The other information has the same interpretation as given for the previous two sections.
- The last section covers the last two bytes of the segment. The section is owned by user 0, The privilege settings are RW and RO for *owner* and *other* respectively. The section is thus protected from other users' write access.

The top of the section window identifies the segment being monitored, as well as providing the segment's section statistics. Specifically, section "CustTable" (Mid = 2) is 8192 bytes in size and has four sections defined on it, two of which are locked. Also included at the top of the section window is the list of users (if any) that are currently blocked trying to read or write data to or from the segment being monitored.

In the example, user 4 is currently blocked attempting a write operation to the segment. The write operation target starts at offset 10 in the segment and extends for 32 bytes. A second user, user 7, is blocked attempting to read 128 bytes starting at offset 1024 in the segment.
Finally, because the monitor is in a trace mode, the next MemSys operation to be executed is reported in the Trace Window. User 14 is about to unlock the section he is holding.

Section Window Commands

MemView commands can be used from within the section window in the same manner that they are used from the main monitor window. Examples:

Command	Effect
i*n*	Set the interval to *n* milliseconds flow mode
ts	Enter trace step mode
w*n*	Open a watch window on segment *n*
s*n*	Open a section window on segment *n*
b*n*	Browse the contents of segment *n*
q	Exit the watch window

Additional commands are available that are specific to the section window (and watch window). They provide a means for scrolling within the section window data. These commands are:

Command	Effect
((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

Scrolling only works where it makes sense. Otherwise, the command is ignored.

3.4.7 Browsing a Shared Memory Segment

Segment browsing is also provided from within MemView. Using this capability, a programmer can verify a segment's data content or search for specific Hex or ASCII memory patterns.
Unlike the windows just described, when the browse facility is used, it temporarily freezes the subject MemSys instance.
Browsing is initiated using the command string "bN," where N is the Mid to be browsed.

Example:
The command to initiate browsing of Mid 5 is:

```
Command> b5
```

The browse facility uses a full screen window for displaying shared memory contents.

Example:

```
Segment:  Mid = 2  Size = 8192  Name = CustTable  Sections = 2/4

            4W(10 32) 7R(1024 128)

 08020:   00000000 00000000 00313030          .........1000011
 08040:  30303131                             Harvey Schmidlap
 08060:   48617276 65792053 63686e69          p 094-40-2819 37
 08080:  646c6170                             Street Lex City.
 08100:   70203039 342d3430 2d323831          07761....23.00..
 08120:  39203337                             .........4311901
 08140:   53747265 6574204c 65782e20          Joe Wilson 072-4
 08160:  43697479                             5-8019 Peteville
 08180:   30373736 31000101 73323347          ............
         30307274
           00000000 00000000 00343331
         31393031
           4a656520 57696c73 6f6e2030
         37322d34
           352d3830 31392050 65746556
         696c6c65
           00000000 00000000 00000000

 Command:
Offset = 0
```

The top line identifies the segment being browsed. Beneath that is the list of users currently blocked writing or reading the segment.
The body of the screen presents the segment's text in hex and ASCII. The format used should be familiar by now. It is the same format as the one used for browsing in QueSys and the watch window in MemSys.

3.4.8 Browse Facility Commands

Navigating in and about shared memory segments is accomplished using the browse facility commands.

Scroll commands are:

Command	Effect
⇑ (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

Scrolling only works where it makes sense. Otherwise the command is ignored.
We will see in the next section that searching for a pattern within a segment can cause the segment to scroll to the offset where the pattern is found.

3.4.8.1 ASCII Pattern Searching

The search commands available within the MemSys browse facility are identical to those available in QueSys, except that they apply to a memory segment instead of messages on a queue.
Forward ASCII pattern searching is executed by specifying a pattern between two '/' characters and hitting return. Backward searches are specified using two '\ 'characters. The second bracket character is not always necessary, as shown in the following examples. Repeat patterns are remembered. The following examples demonstrate these points:

Command	Effect
`//`	Repeat the search
`/`	Same
`\IBM\`	Search backward in the current segment for the ASCII string "IBM"
`\\`	Repeat the search
`\`	Same

3.4.8.2 Hexadecimal Pattern Searching

Searching for Hexadecimal patterns is very similar to ASCII pattern searching. The only differences are that the pattern specified is a Hex string, and that an 'x' is appended to the end of the search command.

`/4f37/x`	Search forward for the hex pattern "4f37" within the segment
`\4f37\x`	Search backward for the hex pattern "4f37" within the segment

3.4.8.3 Switching to Another Segment

Switching to browse another segment is accomplished using the "b *n*" command as described above.
This allows navigation between segments without having to exit the browse facility. This is important, since the entire MemSys instance remains frozen. Exiting the browse facility, however briefly, unfreezes the instance.

3.4.8.4 Exiting the Browse Facility

The browse facility is exited using the "q" command. Once browsing is terminated, the MemSys instance is unfrozen.

Example:

```
Command> q
```

3.4.9 Panning with MemView

Panning within MemView lets the developer observe different portions of the interaction matrix. This is especially useful when a zoom window is open and parts of the matrix are not visible.
All "panning" commands start with 'p'.
Vertical panning (up and down) to observe other users is done by specifying a 'u' (for user) and a Uid to pan to.

Example:

```
Command> pu8
```

The above command scrolls the interaction matrix so that Uid 8 is at the top of the display.
Horizontal panning (right and left) to monitor other segments is accomplished specifying a 'm' (for segment) and an Mid to pan to.

Example:

```
Command> pm4
```

The above command scrolls the interaction matrix so that Mid 4 is the first displayed (left-most).

Example:

```
Command> po
```

The command "po" returns the display to the origin of the activity matrix.

3.4.10 Stopping MemView

MemView monitoring is terminated via the 'q' command.

Example:

```
Command> q
```

Bringing down MemView has no effect on the underlying activities of the MemSys instance. It continues to function unaffected. Any overhead incurred by monitoring is eliminated.

4. THE X-*IPC* SEMAPHORE SYSTEM (SEMSYS)

4.1 SemSys Concepts

Two classes of semaphores are available using SemSys.

- Event semaphores
- Resource semaphores

4.1.1 Event Semaphores

X-*IPC* event semaphores are Boolean in nature and are used for signaling the occurrence of events. Event semaphores are either "set" or "clear." Users can wait for "clear" event semaphores to be "set" by other users. The "setting" of an event semaphore will usually unblock (i.e., wake up) at least one of the users waiting for the event to occur.
We will see that it is possible with X-*IPC* to wait on groups of event semaphores in a variety of ways.

4.1.2 Resource Semaphores

X-*IPC* resource semaphores are numerical devices for enforcing accurate and fair resource access control. Resource semaphores are typically used for limiting the concurrent usage of a resource to some preset level. User programs that wish to access or use the resource attempt to acquire a copy of the designated semaphore, perhaps blocking until a copy of the resource is available. Users that release held copies of the resource may in turn cause the wake-up of users which were previously blocked when trying to acquire the resource. Here, too, we will see that it is possible to attempt to acquire multiple resource semaphores in a variety of ways.

4.1.3 Multiple Semaphore Operations

X-*IPC* SemSys supports operations involving multiple semaphores in a straight-forward manner. Using this capability, it is possible to build sophisticated interprocess synchronization schemes.
It is, for example, easy to design systems that:

- "Block until ANY of 5 events have occurred."
- "Block until ALL of a group of resources are available simultaneously (Atomically)."
- "Block until ALL of a group of resources become available over time (Cumulatively)."
- "Block until ALL of a list of events have occurred."

The specifics for coding such constructs are included below in the sections on SemAcquire() and SemWait().

4.2 SemSys Configuration

As we saw earlier, an X•IPC instance is defined by its configuration (`.cfg`) file. The SemSys section of the configuration file describes the composition and capacity of the instance's SemSys.
Three parameters must be set within the SemSys section of the instance configuration file. Additional operating system specific parameters (if required) are listed in the relevant Platform Notes.

The configuration parameters are:

- **MAX_SEMS**, The maximum number of concurrent semaphores. It should be set based on the requirements of the programs using the I nstance.
- **MAX_USERS,** The maximum number of concurrent users and simultaneous asynchronous operations. It should be set based on 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.
- **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 approximating formula to start with is:

  ```
  MAX_NODES = (MAX_SEMS*2) + (MAX_USERS*4) + (MAX_USERS*MAX_SEMS)
  ```

  Empirical observations via SemView should be made to monitor node usage. Adjustments should follow as necessary.

4.3 SemSys Functions

4.3.1 SemCreate() - Creating a New Semaphore

The first step toward using an X•IPC semaphore within an instance is to create it. As we saw earlier, there are two types of semaphores: Event and Resource. A semaphore's type is specified when the semaphore is created.
SemCreate() takes two arguments:

- The name of the new semaphore
- A value indicating the type of the semaphore to be created

SemCreate() returns the "semaphore id" (Sid) of the newly created semaphore. This value is used as the semaphore's "handle" in all subsequent SemSys function calls that refer to this semaphore.

Example:

```
Sid = SemCreate("CritSectSem", 1);
```

In the above example, the calling user attempts to create a new semaphore having the name `CritSectSem`. The new semaphore will be a resource semaphore having a maximum resource count of one. Such a semaphore could be used to enforce single access to an application's critical section.

Example:

```
Sid = SemCreate("BufferSem", 5);
```

In this example, the calling user is creating a resource semaphore having the name `BufferSem` and a maximum resource count of five. Such a semaphore might be used to control orderly access to a system's five usable buffers.

Example:

```
Sid = SemCreate("NetworkDownSem", SEM_CLEAR);
```

Here the calling process creates an event semaphore with the name `NetworkDownSem`. The semaphore is created with an initial state of "clear." Such a semaphore might be employed to notify user programs within an application that a network has come down. Event semaphore creation differs from resource semaphore creation in the value given for SemCreate()'s second argument. Resource semaphore creation specifies the maximum resource count. Event semaphore creation specifies the semaphore's initial state as `SEM_SET` or `SEM_CLEAR.`
Duplicate semaphore names are *not* allowed within an instance.
Specifying `SEM_PRIVATE` as the name of the new semaphore creates a semaphore inaccessible via SemAccess(), effectively making its Sid private to the creating program. Of course, the creating program can pass the Sid to others, if it so wishes. The advantage of using `SEM_PRIVATE` as a name is that it is guaranteed not to conflict with any semaphore name currently in the instance.

4.3.2 SemAccess() - Accessing an Existing Semaphore

Once a semaphore has been created, other users can access its Sid using SemAccess(). SemAccess() takes one argument:

- ☐ The name of an existing semaphore.

SemAccess() returns the "semaphore id" (Sid) of the desired semaphore. This value is used as the semaphore's "handle" in all subsequent SemSys function calls that refer to the semaphore.

Examples:

```
CritSid    = SemAccess("CritSectSem");
BuffSid    = SemAccess("BufferSem");
NetDownSid = SemAccess("NetworkDownSem");
```

The above example accesses the three semaphores created in the previous section.

4.3.3 SemListXxx() – Manipulating Semaphore Lists

SemSys operations that manipulate semaphores do so using semaphore id lists. Manipulating a single semaphore is accomplished using a list having one element.

A list of Sids is referred to as a SidList. A SIDLIST data type is defined for creating and working with SidLists. Functions expecting a list of Sids as one of their arguments take a SIDLIST data type for this purpose.
There are two functions for building SidLists: SemList() and SemListBuild()
SemList() takes a list of Sids as its arguments with SEM_EOL marking the end of the list. SemList() creates a SidList in its internal static area. For this reason, the returned SidList can be safely used only once.

Example:

```
RetCode = SemRelease(SemList(Sid1, Sid2, Sid3, SEM_EOL), ... );
```

SemRelease() expects a SidList as its first argument. (SemRelease() is described in a later section.) In the above example, SemList() is used "on the fly" to create the SidList argument for SemRelease().
SemListBuild() takes a SIDLIST variable as its first argument. The remaining arguments are a list of Sids as described for SemList(). SemListBuild() creates a SidList in the user-provided SIDLIST variable. This SidList can safely be reused by the programmer.

Example:

```
SIDLIST SidList;
XINT Sid1, Sid2, Sid3;

SemListBuild(SidList, Sid1, Sid2, Sid3, SEM_EOL);
...
RetCode = SemAcquire(SEM_ALL, SidList, NULL, SEM_WAIT);
...
/*
 * Work with resources associated with resource
 * semaphores Sid1, Sid2 and Sid3
 */
...
RetCode = SemRelease(SidList, NULL);
```

SemAcquire(), like SemRelease(), takes a SidList as an argument (SemAcquire() and its SEM_ALL option are described in a later section). The SidList built with SemListBuild() can be used repeatedly.
A SidList must not exceed SEM_LEN_SIDLIST elements. This is usually not a great concern since SEM_LEN_SIDLIST is currently defined to be 32.
Two additional functions, SemListAdd() and SemListRemove(), allow for updating SidLists dynamically, and another function, SemListCount(), allows determination of the number of elements in a SidList.
SemListAdd() is provided to allow the programmer to add SidList elements to an existing SidList (i.e., one that has been created by SemListBuild()). This is a common requirement in situations where the needed SidList must be built dynamically based on certain run-time conditions.
SemListRemove() is provided to allow the programmer to remove SidList elements from an existing SidList when necessary.

The calling sequence for SemListAdd() and for SemListRemove() is identical to that of SemListBuild(). It too expects a user-provided SidList as its first argument. The listed SidList elements are added to or removed from that SidList.

Example:

```
/*
 * A SidList containing Sid1, Sid2 and Sid3 is created
 * one element at a time using SemListAdd(), as follows:
 */

SIDLIST SidList;
XINT Sid1, Sid2, Sid3;
...

SemListBuild(SidList, SEM_EOL);
SemListAdd(SidList, Sid1, SEM_EOL);
SemListAdd(SidList, Sid2, SEM_EOL);
SemListAdd(SidList, Sid3, SEM_EOL);
```

In the following example, SemListRemove() is used so that the resource semaphores Sid1, Sid2 and Sid3 are each acquired (SemAcquire() and its SEM_ANY option are described in a later section), then released, in whatever order they become available:

```
SIDLIST SidList;
XINT Sid1, Sid2, Sid3, AcquiredSemID;

SemListBuild(SidList, Sid1, Sid2, Sid3, SEM_EOL);

/*
 * Acquire and release resource semaphores Sid1, Sid2, and Sid3,
 * one at a time, in whatever order they become available.
 */

while (SemListCount(SidList)>0)
{
    RetCode = SemAcquire(SEM_ANY, SidList, &AcquireSemID, SEM_WAIT);
    ...
    /* Work with resource associated with whichever resource semaphore
     * was acquired (AcquiredSemID), then release it and remove it
     * from the SidList
     */
    ...
    RetCode = SemRelease(SemList(AcquiredSemID, SEM_EOL), NULL);
    SemListRemove(SidList, AcquiredSemID, SEM_EOL);
}
```

4.3.4 SemAcquire() - Acquiring Resource Semaphores

SemAcquire() and SemRelease() are used for manipulating SemSys resource semaphores

.

The maximum semaphore value specified when a resource semaphore is created determines the maximum number of copies of the semaphore that can exist at any one time. Users vying for access to the resource represented by the semaphore do so by acquiring and releasing copies of the semaphore.

A user attempts to attain copies of one or more resource semaphores using SemAcquire(). It subsequently releases held semaphore copies using SemRelease().

SemAcquire() takes four arguments:

- A type code indicating the type of acquire operation to perform.
- A SidList holding a list of Sid copies to acquire.
- A pointer to a variable that gets assigned by SemAcquire(). (This value can be `NULL` if no return value is desired.)
- A blocking option code in case the operation needs to block.

Example:

```
/*
 * Create a critical section semaphore.
 * Then, gain access to the critical section by
 * acquiring the only copy of the semaphore.
 */

CritSid = SemCreate("CritSectSem", 1);

RetCode = SemAcquire(SEM_ALL,
                     SemList(CritSid, SEM_EOL),
                     &RetSid,
                     SEM_WAIT);
```

SemAcquire() attempts to access a list of resource semaphore copies. Semaphore acquisition can occur in one of three ways:

- **`SEM_ANY:`** Acquire any of the semaphore copies listed.
- **`SEM_ALL:`** Acquire all of the semaphore copies listed as they become available (i.e., cumulatively).
- **`SEM_ATOMIC:`** Acquire all of the semaphore copies listed, waiting until all of them are available at the same time (i.e., atomically).

In the above example, the SidList has one element (`CritSid`). It is therefore inconsequential which acquire type is specified, since they are all equivalent when applied to a single element list.

When SemAcquire() succeeds, `RetSid` is returned with the Sid of the last semaphore acquired. For single-semaphore operations, this is not very useful information. For SemAcquire() operations involving multiple semaphores, this information can be important. It is acceptable to specify a `NULL` RetSid argument.

When SemAcquire() fails, and the cause of the failure is related to one of the listed semaphores, `RetSid` is assigned the Sid of the problematic semaphore.

Now consider the following example:

```
/*
 * Create a buffer access semaphore to control
 * access to five available buffers.
 * Then access two copies of the semaphore.
 */

BuffSid = SemCreate("BufferSem", 5);

RetCode = SemAcquire(SEM_ATOMIC,
                     SemList(BuffSid, BuffSid, SEM_EOL),
                     &RetSid,
                      SEM_WAIT);
```

This form of SemAcquire() (using SEM_ATOMIC) succeeds only when all of the listed Sid copies are available at one time. The calling user blocks until this occurs, based on the SEM_WAIT argument. At that time, the process is allowed to proceed, presumably to make use of the buffers that have become available.

If the calling user needed control of the critical section as well as access to two buffers before proceeding, then the SemAcquire() would be coded as:

```
RetCode = SemAcquire(SEM_ALL,
                     SemList(CritSid, BuffSid,
                             BuffSid, SEM_EOL),
                     &RetSid,
                     SEM_WAIT);
```

Note that in this example we have assumed that the listed copies could be acquired cumulatively (via SEM_ALL).
There is no significance to the order of Sid specification within the SidList when employing SEM_ALL or SEM_ATOMIC. Within a SEM_ANY call, the listed Sids are pursued in the order listed.

Example:

```
RetCode = SemAcquire(SEM_ANY,
                     SemList(SidA, SidB, SidC, SEM_EOL),
                     &RetSid,
                     SEM_WAIT);
```

In this example, the Sids are checked for availability in the order specified (SidA, SidB and then SidC). If none are available, blocking occurs and the first of the three copies to become available is acquired. The order of specification is by then no longer relevant. In either case, RetSid is returned with the Sid of the semaphore that was acquired.
The above examples demonstrate SemAcquire() using synchronous blocking options. Asynchronous blocking is also possible by specifying one of the three asynchronous blocking options. Refer to the Advanced Topics chapter for a detailed description of the asynchronous blocking options.

4.3.5 SemRelease() - Releasing Resource Semaphores

The flip side of resource semaphore acquisition is releasing resource semaphores. This is accomplished using SemRelease().Resource semaphorecopies must be released by holding users in order for them to be successfully acquired by other users.

SemRelease() takes two arguments:

- A SidList holding a prepared list of Sid copies to release.
- A pointer to a variable that gets assigned by SemRelease().

Example:

```
/*
 * Release the critical section semaphore, and two
 * copies of the buffer semaphores being held.
 */

RetCode = SemRelease(SemList(CritSid, BuffSid,
                             BuffSid, SEM_EOL),
                     &RetSid);
```

It is, of course, an error to attempt to release a semaphore copy not currently held. Here, too, `RetSid` is returned with the identity of an invalid Sid, if one was encountered. As with SemAcquire(), it is acceptable to specify a `NULL` RetSid argument.

4.3.6 SemSet() - Setting Event Semaphores

Semaphores are often needed for interprocess event synchronization and notification. Event semaphores are ideal for such situations. Event semaphores are Boolean in nature. They are either "set" or "clear" at any point in time.
Users waiting for an event to occur typically block on a "clear" event semaphore associated with the event. A user detecting the event's occurrence then "sets" the semaphore, thus allowing the blocked users to proceed.
The ability to operate on multiple resource semaphores in a single operation, described in the section on SemAcquire(), applies similarly to event semaphores. It is thus possible to "set," "clear" and "wait" on multiple events (via their semaphores) using SidLists.
Event semaphore "setting" is accomplished via SemSet().

SemSet() takes two arguments:

- A SidList holding a list of event Sids to set.
- A pointer to a variable that gets assigned by SemSet().

Example:

```
/*
 * Network is detected to have gone down.
 * Set the Network Down event semaphore.
 */

RetCode = SemSet(SemList(NetDownSid, SEM_EOL), &RetSid);
```

A semaphore remains "set" until it is "cleared." It is often required to "clear" an event semaphore some time after it has been "set" so that the event can be waited on again. This is done using the SemClear() function.
`RetSid` is returned with the identity of an invalid Sid, if one was encountered. As with SemClear(), it is acceptable to specify a `NULL` RetSid argument.

4.3.7 SemClear() - Clearing Event Semaphores

SemClear() is the reverse of SemSet() in that it places the listed semaphores into the "clear" state. SemSet() takes two arguments:

- A SidList holding a list of event Sids to clear.
- A pointer to a variable that gets assigned by SemClear().

Example:

```
/*
 * Network has come back up.
 * Clear the Network Down event semaphore.
 */

RetCode = SemClear(SemList(NetDownSid, SEM_EOL), &RetSid);
```

4.3.8 SemWait() - Waiting on Event Semaphores

When a user wants to block until one or more events have occurred, it issues a SemWait() call with a list of event Sids as one of the arguments.
Occasionally it is important that only one of the users blocked on an event semaphore be allowed to proceed once the semaphore is "set." To assure this form of control, SemWait() includes an option flag indicating that the "set" semaphore(s) should be "cleared" once the SemWait() request has been fully satisfied. In effect, the user is given the option of "shutting the door behind him."
SemWait() takes four arguments:

- A type code indicating the type of wait operation to perform.
- A SidList holding a list of event Sids to wait on.
- A pointer to a variable that gets assigned by SemWait(). (This value can be `NULL` if no return value is desired.)
- A blocking option specifying the action to be taken in case the SemWait() needs to block. An optional `SEM_CLEAR` flag may be logically ORed to the left of the specified blocking option, indicating that "set" semaphores should be "cleared" once the SemWait() request has been fully satisfied. An example of this will be provided shortly.

Example:

```
/*
 * Create an event semaphore that will be set when
 * the database is full. The calling user then blocks
 * (via SemWait()) until some other user detects the
 * condition and sets the semaphore. The semaphore
 * remains set after SemWait() returns.
 */

DBFullSid = SemCreate("DatabaseFullSem", SEM_CLEAR_xe "SEM_CLEAR"_);

RetCode = SemWait(SEM_ALL,
                  SemList(DBFullSid, SEM_EOL),
                  &RetSid,
                  SEM_WAIT);
```

SemWait() attempts to wait on a list of event semaphores. Semaphore waiting can take one of three forms:

- **SEM_ANY**: Wait for any of the listed semaphores to be in the "set" state.
- **SEM_ALL:** Wait until all of the listed semaphores have been in the "set" state at least once since the start of the SemWait() operation. All semaphores must have been in the "set" state once, but they are not required to stay "set."
- **SEM_ATOMIC:** Wait until all of the listed semaphores are concurrently in the "set" state (i.e., atomically).

In the above example, the SidList has one element (DBFullSid). It is therefore inconsequential which wait type is specified, as they are all equivalent when applied to a single-element list.

When SemWait() succeeds, `RetSid` is returned with the Sid of the last semaphore waited for. For single-semaphore operations, this is not very useful information. For SemWait() operations involving multiple semaphores, this information can be important. It is acceptable to specify a `NULL` RetSid argument.

When SemWait() fails and the cause of the failure is related to one of the listed semaphores, "RetSid" is assigned the Sid of the problematic semaphore.

Now consider the following example:

```
/*
 * Block until any of the following crisis situations arise:
 * Excessive temperature, humidity or pressure. Clear the set
 * semaphore after the SemWait() operation completes.
 */

RetCode = SemWait(SEM_ANY,
                  SemList(TempSid, HumidSid,
                          PressureSid, SEM_EOL),
                  &RetSid,
                  SEM_CLEAR | SEM_WAIT);

if (RetCode >= 0)    /* Success */
{
    /*
     * SemWait has assigned to "RetSid" the Sid
     * of the event semaphore that was set.
     */

    switch(RetSid)
    {
        case TempSid:
            /* react to excessive Temperature */
            ...
        case HumidSid:
            /* react to excessive Humidity */
            ...
        case PressureSid:
            /* react to excessive Pressure */
            ...
    }
}
```

In the above example, we see how RetSid can be used for identifying and reacting to the event that has occurred. RetSid is also used to determine the identity of an invalid Sid (e.g., a nonexistent Sid: RetCode = SEM_ER_BADSID) if one is encountered. Note that the SEM_CLEAR option flag, when specified, must be ORed to the left of whatever blocking option is designated.

Example:

```
/*
 * Create two event semaphores that indicate network and
 * database startup status. Wait 60 seconds for the two
 * semaphores to be in the "set" state concurrently
 * indicating that they both have come up and are active.
 */

NetSid = SemCreate("NetworkUpSem", SEM_CLEAR);
DatabaseSid = SemCreate("DatabaseUpSem", SEM_CLEAR);

RetCode = SemWait(SEM_ATOMIC,
                  SemList(NetSid, DatabaseSid, SEM_EOL),
                  &RetSid,
                  SEM_TIMEOUT(60));

if (RetCode >= 0)             /* Success */



{
    /*
     * Database and Network are up.
     */
    ...
}
else
if (RetCode == SEM_ER_TIMEOUT)
{
    /*
     * Database and Network have not come up together.
     */
    ...
}
```

This form of SemWait(SEM_ATOMIC) succeeds only when all of the listed event semaphores are concurrently in the "set" state. Based on the 60 second timeout value, the calling user blocks for a maximum of 60 seconds, waiting for this scenario to occur. If the SemWait() does not succeed within 60 seconds, the function return's the SEM_ER_TIMEOUT error code. The program can then react accordingly.
When it is important to react to a certain set of events that occur over a period of time, a user can be started to wait cumulatively for the list of corresponding event semaphores. This would be done using the SEM_ALL wait type.

Example:

```
/*
 * Create event semaphores that are "set" when the machine
 * components they represent are down and otherwise are
 * "clear." Then block and wait until all three components
 * have failed (been in the "set" state) at least once.
 */

Eng1Sid  = SemCreate("Engine1Sem", SEM_CLEAR);
Eng2Sid  = SemCreate("Engine2Sem", SEM_CLEAR);
HydroSid = SemCreate("HydrolicSem", SEM_CLEAR);

RetCode = SemWait(SEM_ALL,
                  SemList(Eng1Sid Eng2Sid,
                          HydroSid, SEM_EOL),
                  &RetSid,
                  SEM_WAIT);
/*
 * React to the unreliable equipment.
 */
...
```

The above examples demonstrate SemWait() using synchronous blocking options. Asynchronous blocking is also possible by specifying one of the three asynchronous blocking options. Refer to the Advanced Topics chapter for a detailed description of the asynchronous blocking options.

4.3.9 SemCancel() - Cancel Blocked SemSys Operations

The SemCancel() function is useful for cancelling blocked SemAcquire() or SemWait() operations involving specific semaphores.
This might be necessary if it has been determined that a blocked request can no longer be satisfied. For example, consider a user that has blocked on acquiring three resources, and one of the resources has become disabled and is no longer in service. Issuing a SemCancel() on that resource semaphore interrupts any and all users blocked on SemAcquire() calls trying to access the no longer available resource.

SemCancel() takes two arguments:

- A SidList holding a prepared list of Sids to cancel blocking on.
- A pointer to a variable that gets assigned by SemCancel().

Example:

```
/*
 * Cancel any blocked SemAcquire()
 * operations involving a resource Sid.
 */

RetCode = SemCancel(SemList(ResourceSid, SEM_EOL), &RetSid);
```

SemCancel() can also be used to cancel SemWait() operations involving event semaphores that will never be "set."
SemCancel()'s "RetSid" is returned with the identity of an invalid Sid, if one is encountered. It is acceptable to specify a `NULL` RetSid argument.

SemAcquire() and SemWait() calls that are cancelled return with `RetCode = SEM_ER_CANCELLED`, and have their `RetSid` assigned to the Sid of the cancelled semaphore.

4.3.10 SemDelete() - Deleting a Semaphore

A semaphore should be deleted from its instance when it is no longer needed. This recycles internal SemSys resources and makes the SemView monitor less cluttered.

SemDelete() takes one argument:

- ☐ The Sid of the semaphore to be deleted.

Example:

```
RetCode = SemDelete(Sid);
```

SemDelete() will succeed only if the subject semaphore is completely inactive at the time. Resource semaphores that have copies "held" by users or that are being waited on cannot be deleted. Event semaphores that are being blocked cannot be deleted.
If a semaphore must be removed regardless of its current status, then SemDestroy() should be employed.

4.3.11 SemDestroy() - Destroying a Semaphore

A semaphore that must be removed from its instance can be destroyed using SemDestroy(). SemDestroy() removes the subject semaphore regardless of the semaphore's current status.

SemDestroy() takes one argument:

- ☐ The Sid of the semaphore to be destroyed.

Example:

```
RetCode = SemDestroy(Sid);
```

When a semaphore is destroyed, a number of things happen:

- ☐ All SemAcquire() or SemWait() operations involving the destroyed semaphore are cancelled and returned with `RetCode = SEM_ER_DESTROYED`. Their `RetSid` is assigned the Sid of the destroyed semaphore.
- ☐ All users holding one or more copies of a destroyed resource semaphore have these copies removed from their ownership. This occurs silently and it is the responsibility of the program to adjust to the semaphore's destruction.

For obvious reasons, SemDestroy() should be used sparingly. Its most likely application would be within the execution of a system's "cleanup" program at which time the above side-effects are normally of no concern.

4.3.12 SemInfoSys() - Information about an Instance's SemSys

X•IPC provides a set of SemSys functions that can be used to access status information about various aspects of an instance's SemSys.
The returned data can be used to make run-time decisions about ongoing application processing.

SemInfoSys() returns with information about the instance's SemSys that the user is logged into. SemInfoSys() takes one argument:

- A pointer to a SEMINFOSYS structure that is returned filled with the subsystem's status information.

Example:

```
SEMINFOSYS SysData;

RetCode = SemInfoSys(&SysData);
```

A complete description of how to use the Info functions is presented in the Advanced Topics chapter of the X•IPC User Guide, in the section entitled, "Info Function List Manipulation."
The definition of the SEMINFOSYS datatype is included in the User Data Structures chapter of the QueSys/MemSys/SemSys Reference Manual.

4.3.13 SemInfoUser() - Information about a SemSys User

SemInfoUser() returns with information about a specified user. SemInfoUser() takes two arguments:

- The Uid whose status is desired.
- A pointer to a SEMINFOUSER structure that is returned filled with the user's status information.

Besides statistical data, the SEMINFOUSER structure returns with "list" data related to the specified user. Each user has an *HList*, *QList* and *Wlist* associated with it.

- The HList is the list of resource Sid copies currently held by the subject user. The Sids are listed in the order in which they were acquired.
- The QList is 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.
- The WList is 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 lists within SEMINFOUSER are arrays that can accommodate up to SEM_LEN_INFOLIST elements. The actual lists may, at times, be greater than SEM_LEN_INFOLIST elements in length. A call to the SemInfoUser() function must therefore be preceded by the setting of three SEMINFOUSER structure members (*HListOffset*, *QListOffset* and *WListOffset*) with values specifying what portions of the three respective lists are desired.
More specifically, before SemInfoUser() is called, the three list offset variables within the SEMINFOUSER structure must be set, indicating from what point in each list to return data. Setting the offsets to zero directs the function to return with list data from the start of the lists.

Example:

```
SEMINFOUSER UserData;

UserData.HListOffset = 0;
UserData.QListOffset = 0;
UserData.WListOffset = 0;

RetCode = SemInfoUser(Uid, &UserData);
```

A complete description of how to use the Info functions is presented in the Advanced Topics chapter of the *X•IPC* User Guide, in the section entitled, "Info Function List Manipulation."
The definition of the SEMINFOSYS datatype is included in the User Data Structures chapter of the QueSys/MemSys/SemSys Reference Manual.

4.3.14 SemInfoSem() - Information about a SemSys Semaphore

SemInfoSem() returns with information about a specified semaphore. SemInfoSem() takes two arguments:

- The Sid whose status is desired.
- A pointer to a SEMINFOSEM structurethat is returned filled with the semaphore's status information.

Besides statistical data, the SEMINFOSEM structure returns with "list" data related to the specified semaphore. Each semaphore has an *HList* and *WList* associated with it:

- The HList is the list of Uids currently holding copies of the subject resource semaphore. The HList of event semaphores is always empty. The Uids are listed in the order that they acquired the semaphore copies.
- The WList is the list of Uids currently waiting on the subject semaphore. The Uids are listed in the order that they began waiting.

The lists within SEMINFOSEM are arrays that can accommodate up to SEM_LEN_INFOLIST elements. The actual lists may, at times, be greater than SEM_LEN_INFOLIST elements in length. A call to the SemInfoSem() function must therefore be preceded by the setting of two SEMINFOSEM structure members (*HListOffset* and *WListOffset*) with values specifying what portions of the two respective lists are desired.
More specifically, before SemInfoSem() is called, the two list offset variables within the SEMINFOSEM structure must be set, indicating from what point in each list to return data. Setting the offsets to zero directs the function to return with list data from the start of the lists.

Example:

```
SEMINFOSEM SemData;

SemData.HListOffset = 0;
SemData.WListOffset = 0;

RetCode = SemInfoSem(Sid, &SemData);
```

A complete description of how to use the Info functions is presented in the Advanced Topics chapter of the X•IPC User Guide, in the section entitled, "Info Function List Manipulation."
The definition of the SEMINFOSEM datatype is included in the User Data Structures chapter of the QueSys/MemSys/SemSys Reference Manual.

4.3.15 SemFreeze() - Freezing SemSys

X•IPC provides the user with the ability to attain complete and exclusive control over the SemSys subsystem of an instance.
This mechanism allows a user to execute a series of SemSys operations, with the assurance that no other user's SemSys operations are interwoven with his.
Such a capability is of particular importance when a user requires a complete and consistent view of the state of activity occurring within SemSys. With it, multiple SemInfoXxx function calls can be executed for collecting status data with the guarantee that the subsystem's state is unchanged between the SemInfoXxx calls.
SemFreeze() takes no arguments.

Example:

```
/*
 * Produce SemSys snapshot status report.
 */

SemFreeze();

/*
 * Collect the data.
 */
...
SemInfoSys(...);
...
SemInfoSem(...);
...
SemInfoUser(...);

/*
 * Unfreeze SemSys and report results.
 */

SemUnfreeze();

printf(...);
```

A further note regarding SemFreeze(): It is an error for a user to issue a blocking SemSys function call, specifying a blocking option code (i.e., SEM_WAIT or SEM_TIMEOUT), once the user has frozen the subsystem.

4.3.16 SemUnfreeze() - Unfreezing SemSys

SemUnfreeze() is the bracketing function to SemFreeze(). It returns the SemSys subsystem to its unfrozen state. Other SemSys users resume normal SemSys operations.

Example:

```
SemUnfreeze();
```

SemUnfreeze() will fail if the calling user has not frozen the subsystem.

4.4 The SemSys On-Line Monitor: SemView

SemView is the on-line monitor for *X•IPC* SemSys.

4.4.1 Starting SemView

SemView is started from the command line using the SemView command.

SemView takes two arguments:

- The first argument is the initial "interval" snapshot setting. It defines in milliseconds the initial update frequency of the monitor. The interval argument is mandatory.
- The second argument is the instance file name of the instance to be monitored. This argument is optional. If it is omitted, SemView uses the value of the `xipc` environment variable for the instance file name of the instance to start monitoring.

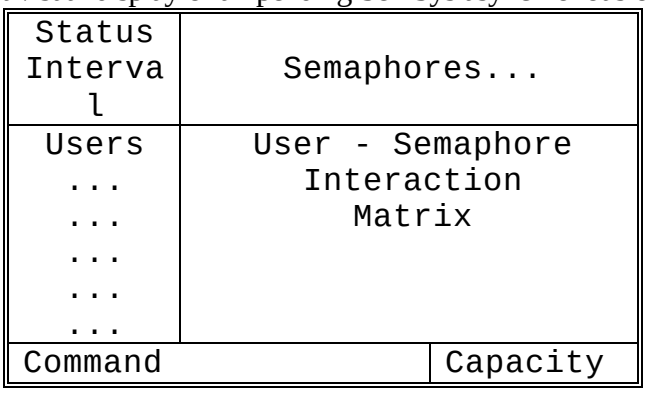
Example:

```
semview 100 /usr/demo
```

The above command starts the SemView monitor for the SemSys subsystem of the /usr/demo instance. The initial update interval is set to 100 milliseconds.

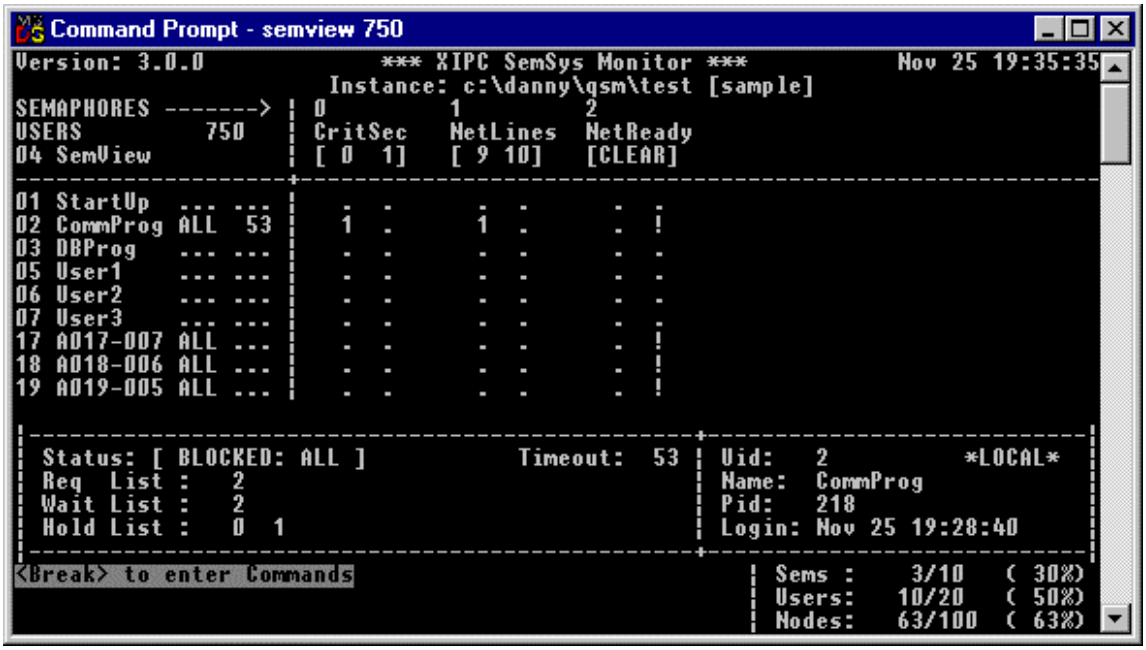
4.4.2 SemView Layout

SemView's main display is matrix-like in appearance. Users logged into the instance and existing SemSys semaphores form the axes of the matrix. Interaction between instance users and semaphores is displayed within the body of the "interaction matrix."
SemSys operations that block asynchronously are treated as pseudo-users of SemSys. These *Asynchronous Users* are displayed in the same manner as ordinary users, thus providing a consistent visual display of all pending SemSys asynchronous operations.



Monitor status and interval setting are shown at the top left portion of the screen. SemSys capacity data is displayed at the lower right portion of the screen. The command entry window is at the lower left of the screen.

4.4.2.1 Sample SemView Screen Display



4.4.2.2 User Entries

Users logged into the instance are listed on the left side of the interaction matrix, one line per user.

Each user entry includes:

- The user's SemSys user ID
- The user's login name
- The user's blocking status (if any)
- The blocking timeout value (if any)

An example (not associated with the screen presented above) follows.

```
02 StartUp
 03 CommProg ANY
 06 DBProg   ALL  24
 39 A039-003 ATM
```

In this example, four SemSys users are identified--three real users and one asynchronously blocked operation.

- SemSys user 2 has the login name `StartUp`. The user is not blocked on any SemSys operation.
- SemSys user 3 has logged in as `CommProg`. He is blocked on a SemAcquire() or SemWait()operation involving `SEM_ANY`. He is blocked indefinitely and thus has no timeout value.

- SemSys user 6 has logged in as `DBProg`. He is blocked on a cumulative all (`SEM_ALL`) operation and has a timeout pending with 24 seconds remaining on the blockage.
- SemSys user 39 is a pending asynchronous operation. It is blocked on an atomic all (`SEM_ATOMIC`) operation. The user name `ASYNC-03` is assigned to the asynchronous operation to indicate that the operation was initiated by user 3.

4.4.2.3 Semaphore Entries

The instance's semaphores are identified across the top of the interaction matrix.

Each semaphore entry includes:

- The Sid of the semaphore.
- The user-assigned ASCII name of the semaphore.
- The semaphore's current status: for resource semaphores, it is the currently available and the maximum number of copies; for event semaphores, it is the current state of the semaphore (`CLEAR` or `SET`).

An example (not associated with the screen presented above) follows.

```
  0           1          4
CritSect    Buffers    NetUp
[ 0  1 ]    [ 3  10]   [ SET ] ...
```

In this example:

- Semaphore `CritSect` is shown to have an Sid of 0. It is a resource semaphore with a maximum copy count of 1 and a currently available copy count of 0 (i.e., a user is holding the critical section).
- Semaphore `Buffers` has Sid 1. It has current and maximum copy counts of 3 and 10 respectively (i.e., 7 buffers are currently checked out).
- Semaphore `NetUp` has Sid 4. It is an event semaphore. It is currently "set."

4.4.2.4 Interaction Matrix Cells

Each cell on the SemView interaction matrix describes the current relationship between a user and a semaphore. The exact information appearing in the cell depends on the type of semaphore involved.

For resource semaphores, the cell reports:

- The number of copies of the resource semaphore held by the user.
- The number of copies of the resource semaphore requested by the user.

Examples:

A cell value of " 3 1 " indicates that the user holds three copies of the resource semaphore and is currently blocked attempting to acquire an additional copy.

A cell containing " . 2 " means that the user does not hold any copies of the semaphore and is attempting to acquire two.

A cell containing " 1 . " means that the user holds one copy of the semaphore and is not attempting to acquire any more.

A cell containing " . . " means that the user isn't holding or attempting to acquire any copies of the semaphore.

For event semaphores, the cell reports:

? Whether or not the user is blocked on the semaphore, i.e., waiting for the event semaphore to become "set."

Examples:

A cell having " ! " (an exclamation mark) means that the user is waiting for the semaphore to be "set."

A cell with " . . " means that the user is NOT waiting for the semaphore to be "set."

4.4.3 Monitoring Modes

The topic of monitoring modes–the available options and when they should be used–is described in detail in the X•IPC User Guide.

4.4.4 SemView Zoom Windows

SemView provides the developer with two zoom capabilities.

4.4.4.1 Zooming in on a User

The SemView user zoom window creates a detailed display of the status of a particular SemSys user. The command string for user zooming is `zuN` where `N` is the Uid to be zoomed in on.

Example:

The command for opening a zoom window on the user having a Uid of 4 is:

```
Command> zu4
```

```
Status:  [ NOT BLOCKED ]          |  Uid: 4
  Req  List :                     |  Name: DownLoad
  Wait List :                     |  Pid:  1023
  Hold List :  2  2  4            |  Login: Dec 23
                                  |  12:23
```

User `DownLoad` holds two copies of the resource Sid 2 and one copy of resource Sid 4. The user is otherwise not currently blocked on any SemSys semaphores.

```
Status:  [ BLOCKED: ALL ]         |  Uid: 4
  Req  List :  0  1               |  Name: DownLoad
  Wait List :  0  1               |  Pid:  1023
  Hold List :  2  2  4            |  Login: Dec 23
                                  |  12:23
```

The user is now blocked acquiring a copy of semaphore Sid 0 and Sid 1 (i.e., `ALL`).

```
Status:  [ BLOCKED: ALL ]          |  Uid: 4
  Req  List :  0  1                |  Name: DownLoad
  Wait List :  1                   |  Pid:  1023
  Hold List :  2  2  4  0          |  Login: Dec 23
                                   |  12:23
```

Sid 0 has become available and was acquired. The user continues to block for Sid 1.

Note that the original request is reported as `Req List:`, and the outstanding part of the request is identified as Wait List:.

```
Status:  [ NOT BLOCKED ]           |  Uid: 4
  Req  List :                      |  Name: DownLoad
  Wait List :                      |  Pid:  1023
  Hold List :  2  2  4  0  1       |  Login: Dec 23
                                   |  12:23
```

The request has been satisfied and the user is no longer blocked.

4.4.4.2 Zooming in on a Semaphore

It is also possible to zoom in on a SemSys semaphore. The information contained in the zoom window depends on the type of semaphore being watched.
The SemView semaphore zoom window creates a detailed display of the status of a particular SemSys semaphore. The command string for semaphore zooming is `zsN` where `N` is the Sid to be zoomed on.

Example:

The command for opening a zoom window on the semaphore having Sid of 6 is:

```
Command> zs6
```

If Sid 6 is a resource semaphore, the zoom window will appear as:

```
Maximum Value: 5     Current Value: 1  |  Sid: 6  [Resource]
  Last Uid  :  3                       |  Name: BufferSem
  Wait List :                          |  CreateUid: 2
  Hold List :  2  2  1  3              |  Login: Dec 23 12:23
```

BufferSem is a resource semaphore with a maximum copy value of five. It currently has only one copy available. The remaining four copies are held by Uid 2 (holding two copies), Uid 1 and Uid 3. There are currently no users blocked on the semaphore. The `Last Uid:` field identifies the user that most recently acquired or released the semaphore.

```
Maximum Value: 5     Current Value: 0  |  Sid: 6  [Resource]
  Last Uid  :  4                       |  Name: BufferSem
  Wait List :  6                       |  CreateUid: 2
  Hold List :  2  2  1  3  4           |  Login: Dec 23 12:23
```

Uid 4 has acquired the last copy of Sid 4. Uid 6 has also attempted to access a semaphore copy and is currently blocked.

If Sid 7 is an event semaphore, the zoom window will appear as:

```
Status:  [ CLEAR ]                    Sid: 7  [Event]
  Last Uid  :  1                      Name: RadiationLeak
  Wait List :  4  2  12  1            CreateUid: 13
                                      Login: Dec 23 12:23
```

The event semaphore window is very similar to the resource zoom window. The main difference is that there is no `Hold List:` line. That is because event semaphores are not held.
Here, event semaphore `RadiationLeak` is being monitored by four Uids (i.e., users). If a radiation leak is detected, prescribed steps will be taken (e.g., containing the accident, shutting access to the accident's location, notifying plant personnel, etc.).

4.4.5 Panning within SemView

Panning within SemView lets the developer observe different sections of the interaction matrix. This is especially useful when a zoom window is open and parts of the matrix are not visible.
All panning commands start with `p`.
Vertical panning (up and down) to observe other users is done by specifying a `u` (for user) and a Uid to pan to.

Example:

```
Command> pu8
```

The above command scrolls the interaction matrix so that Uid 8 is at the top of the display.
Horizontal panning (right and left) to monitor other semaphores is accomplished by specifying an `s` (for semaphore), and a Sid to pan to.

Example:

```
Command> ps4
```

The above command scrolls the interaction matrix so that Sid 4 is the first displayed (left-most).

Example:

```
Command> po
```

The command `po` returns the display to the origin of the interaction matrix.

4.4.6 Stopping SemView

SemView monitoring is terminated via the `q` command.

Example:

```
Command> q
```

Bringing down SemView has no effect on the underlying activities of the SemSys instance. It continues to function unaffected. Any overhead incurred by monitoring is eliminated.

5. ADVANCED TOPICS

5.1 Asynchronous Operations

5.1.1 Introduction

X•IPC operations that can block can complete synchronously or asynchronously. The `WAIT` and `TIMEOUT` synchronous blocking options actually block the program that initiated the *X•IPC* operation until the operation completes–either successfully or in failure–at which time the program becomes unblocked and continues its processing.
X•IPC asynchronous options provide a more powerful set of alternatives. Unlike the synchronous options, asynchronous options indicate that the subject *X•IPC* operation should complete in the background, without blocking the calling program. The program is allowed to proceed. When the operation completes, some form of notification is given by *X•IPC*, depending on the asynchronous option specified at the start of the operation.
X•IPC supports three asynchronous options. Each describes a different form of notification to be given by *X•IPC* at the completion of the operation.

- The `CALLBACK` option directs *X•IPC* to execute a user-specified callback function upon completion.
- The `POST` option directs *X•IPC* to set a SemSys event semaphore when the operation completes.
- The `IGNORE` option directs *X•IPC* to allow the operation to complete "silently" with no explicit form of notification.

The three options are described in more detail below. An operation that is invoked asynchronously returns the `MOM_ER_ASYNC, QUE_ER_ASYNC` return code as appropriate.

5.1.2 The Asyncresult Control Block (ACB)

Tracking of an asynchronous *X•IPC* operation is achieved using an Asynchronous Result Control Block (ACB). An ACB is a user-declared structure (of type `ASYNCRESULT`) that is associated with an asynchronous *X•IPC* operation. Each *X•IPC* operation that is coded with an asynchronous blocking option is required to specify an ACB (actually, a pointer to an ACB) along with the option. (Examples are provided below.) The ACB is the vehicle by which *X•IPC* transmits return data when the operation completes.
An ACB also contains a number of fields that support the tracking of asynchronous operations while they are still pending.
When an *X•IPC* operation executes asynchronously, the operation's ACB is set with information for tracking the operation.

- An asynchronously blocked operation is treated as a pseudo-user within the subsystem that it blocked. As such, the pending operation is recorded as an entry in the subsystem's user table and is assigned its own User ID–or, more precisely, an Asynchronous User Id (AUid). The **`AUid`** field of the ACB is set with the blocked operation's AUid.

User information functions that accept a Uid as an argument, such as SemInfoUser(), accept an AUid as well. X•IPC 's subsystem monitors present status on AUid's in the same manner as for ordinary Uid's. This provides the developer with the means for tracking all pending asynchronous operations occurring within an instance, *without* having to "invent" specialized async monitoring tools.

Asynchronous operations that succeed without blocking have the AUid field of their associated ACB set to zero.

- The **`AsyncStatus`** field remains set as `XIPC_ASYNC_INPROGRESS` as long as the operation is pending completion. When the operation completes, the field is set to `XIPC_ASYNC_COMPLETED`. This is most useful for asynchronous operations started with the `IGNORE` option. In that case, the AsyncStatus field being set to `XIPC_ASYNC_COMPLETED` is the only direct indication given by X•IPC that the operation has completed.
- The **`User Data`** fields are useful for passing application information between the point where the asynchronous operation is initiated, and the logic that handles its notification of completion. The information passed is application dependent.
- The **`OpCode`** field is set to the appropriate `XIPC_OPCODE_APINAME` macro value that identifies the X•IPC function call associated with the ACB. Examples include `XIPC_OPCODE_SEMWAIT`, `XIPC_OPCODE_QUESEND`, etc.

The remaining elements within the ACB are a union of structures, one structure per blockable X•IPC API. The appropriate structure is set with return data from the completing asynchronous operation with which it is associated.
The ASYNCRESULT structure is defined as:

```
/*
 * 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;    /* of completed async operation */
        }
        MomEvent

    }
    Api;

};
```

5.1.3 ACB Return Values

The results of an asynchronously blocked operation are returned within the ACB of the completed operation. The one important exception to this is the treatment of what can be generalized as "text data."

When an X•IPC operation that specifies a text buffer as an argument blocks asynchronously and then subsequently completes, the originally specified user text buffer is used when the operation completes. So, for example, a completing QueReceive() operation receives data into the text data buffer that was specified when the QueReceive() was initially called. This is true for all of the X•IPC functions that manipulate "text data." They are: MomSend(), MomReceive(), QueWrite(), QueSend(), QueReceive(), MemWrite() and MemRead().

It is therefore a dangerous practice to pass stack space variables as text data arguments to asynchronously blocking X•IPC functions calls. Static or heap storage variables should be used instead.

5.1.4 The Callback Option

The CALLBACK option directs X•IPC to interrupt the calling program when the asynchronously blocked operation completes by having it execute a user specified callback function. This form of completion notification is the most severe in terms of "rudeness" and should be used in situations where the indicated urgency is called for. Example:

```
/*
 * Wait for any one of three event semaphores to become set.
 * A callback function will execute when the operation completes.
 */

ASYNCRESULT Acb;
VOID        Funct();
XINT        RetSid;
XINT        RetCode;


RetCode = SemWait ( SEM_ANY,
                    SemList(Sid1, Sid2, Sid3, SEM_EOL),
                    &RetSid,
                    SEM_CALLBACK(Funct, &Acb)
                  );

if (RetCode == SEM_ER_ASYNC)
{
    /*
     * Operation executing asynchronously.
     */

    printf ("SemWait executing asynchronously, AUid = %d\n",
          Acb.AUid );
}
else
{
    /*
     * Error !!
     */
}
...
...


VOID
Funct (Acb)
ASYNCRESULT *Acb;
{
    printf ("SemWait completed.\n");
    printf ("RetCode = %d\n", Acb->Api.SemWait.RetCode);
    printf ("RetSid  = %d\n", Acb->Api.SemWait.RetSid);
    ...
}
```

Because it is sometimes important that an operation return *synchronously* if it can complete without blocking, resort to the asynchronous option *only* when the operation cannot immediately complete.
The above example could be modified as follows to force such behavior:

```
/*
 * Wait for any one of three events semaphores to become set.
 * Block asynchronously, if necessary. Otherwise, return
 * immediately with the operation's result.
 */
```

```
ASYNCRESULT Acb;
VOID        Funct();
XINT        RetSid;
XINT        RetCode;

RetCode = SemWait ( SEM_ANY,
            SemList(Sid1, Sid2, Sid3, SEM_EOL),
            &RetSid,
            SEM_RETURN | SEM_CALLBACK(Funct, &Acb)
            );


if (RetCode == SEM_ER_ASYNC)
{
    /*
     * Operation blocked asynchronously.
     */

    printf ("SemWait blocked asynchronously, AUid = %d\n",
          Acb.AUid );
}
else
{
    /*
     * Operation completed immediately. Process results in-line.
     */
    ...
    ...
}
...
...


VOID
Funct (Acb)
ASYNCRESULT *Acb;
{
    printf ("SemWait completed.\n");
    printf ("RetCode = %d\n", Acb->Api.SemWait.RetCode);
    printf ("RetSid  = %d\n", Acb->Api.SemWait.RetSid);
    ...
}
```

It is often convenient to have a single callback function serve multiple pending asynchronous operations. The application could then use the various ACB User Data fields to discern between the pending operations as they complete. One option: assign an identifying code to each ACB, using one of the User Data fields.

5.1.5 The Post Option

The POST option directs X•IPC to set the specified SemSys event semaphore upon completion of the specified operation. This form of completion notification is less intrusive than the CALLBACK option in that no program is directly interrupted as a result of the operation's completion.
Example:

```
/*
 * Receive message having Priority = 100.
 * Semaphore "PostSid" is to be set when the message is received.
 */
```

```
RetCode = QueReceive ( QUE_Q_ANY,
              QueList( QUE_M_PREQ(Qid1, 100), QUE_EOL),
              MsgBuf,
              MsgLen,
              &RetPrio,
              &RetQid,
              QUE_POST(PostSid, &Acb)
            );


if (RetCode == QUE_ER_ASYNC)
{
    /*
     * Operation executing asynchronously.
     */

    printf ("QueReceive executing asynchronously, AUid = %d\n",
          Acb.AUid );
}
    ...
    ...
}
else
{
    /*
     * Error !!
     */
}
```

This example may also be modified to return *synchronously* if the operation succeeds without blocking:

```
/*
 * Receive message having Priority = 100. Block
 * asynchronously if necessary.  Otherwise, operation
 * results are returned immediately.
 */

RetCode = QueReceive ( QUE_Q_ANY,
               QueList( QUE_M_PREQ(Qid1, 100), QUE_EOL),
               MsgBuf,
               MsgLen,
               &RetPrio,
               &RetQid,
               QUE_RETURN | QUE_POST(PostSid, &Acb)
             );

if (RetCode == QUE_ER_ASYNC)
{
    /*
     * Operation blocked asynchronously.
     */

    printf ("QueReceive blocked asynchronously, AUid = %d\n",
          Acb.AUid );
}
else
{
    /*
     * Operation Completed immediately.  Process results in-line.
     */
    ...
    ...
}
```

Reacting to a completed asynchronous operation that specified the POST option can be handled by the original calling program at some later point in its logic when it is convenient for it to issue a SemWait() call regarding the post semaphore, or possibly by a second program waiting for the post semaphore to become set.
In fact, the wait for the post semaphore can be asynchronous as well. It is plain to see how a domino-effect can very easily be created between processes.

5.1.6 The Ignore Option

The IGNORE option directs *X•IPC* to complete the subject operation silently, if it blocks asynchronously. This form of notification is the most passive of the asynchronous options in that no explicit notice of the operation's completion is given by *X•IPC*. The ACB's *AsyncStatus* field is set to XIPC_ASYNC_COMPLETED when the operation it represents completes. The field may be examined periodically to determine when this has occurred. Consider the following example: If segment Mid is locked at the time of the MemWrite() calls, then the two operations will remain pending asynchronously until the segment is unlocked and the MemWrite() operations are permitted to complete. No explicit notice is given by *X•IPC* when the operations complete. The two ACB's can be examined later to confirm completion.
Example:

```
/*
 * Write two records to a shared memory table.
 * The operations complete silently in the background.
 */

XINT      Mid, Offset1, Offset2;
XINT      Size1, Size2, RetCode;
ASYNCRESULT   Acb1, Acb2;


RetCode = MemWrite (Mid, Offset1, Size1, MEM_IGNORE(&Acb1) );

if (RetCode != MEM_ER_ASYNC)
    /*
     * Error !!
     */
...
...


RetCode = MemWrite (Mid, Offset2, Size2, MEM_IGNORE(&Acb2) );

if (RetCode != MEM_ER_ASYNC)
    /*
     * Error !!
     */

...
...
```

Here again the MemWrite() function calls could have been coded to return synchronously, if they complete without blocking, by specifying the `MEM_RETURN` flag logically ORed with the `MEM_IGNORE` option.
Example:

```
RetCode = MemWrite(..., MEM_RETURN | MEM_IGNORE(...));
```

5.1.7 Aborting a Pending Asynchronous Operation

It is occasionally necessary for a program to abort a pending asynchronous operation before it completes. The functions MomAbortAsync(), QueAbortAsync(), SemAbortAsync() and MemAbortAsync() can be used to cancel blocked asynchronous operations in their respective subsystems.

The functions take one argument:

- The AUid of the asynchronous operation to abort.

Example:

```
if ( SemWait( SEM_ANY,
          SidList,
          &RetSid,
          SEM_IGNORE(&Acb)) == SEM_ER_ASYNC)
{
    /*
     * Do other work ...
     */

    ...
    ...

    /*
     * If operation is still pending, then
     * abort it.
     */

    if (Acb.AsyncStatus == XIPC_ASYNC_INPROGRESS)
        SemAbortAsync(Acb.AUid);
}
```

5.1.8 Mixing Asynchronous and Synchronous Operations

The current version of X•IPC employs an interrupt mechanism for implementing asynchronous functionality on most of its supported platforms. Exceptions include MS-Windows 3.x, Windows NT and X-Windows. This means that a process that issues an X•IPC asynchronous operation must be prepared to be *silently* interrupted by X•IPC when the operation completes. At that time X•IPC internally reacts to the operation's completion. This is an important consideration if the process can block synchronously at points within its logic. Calls to such synchronous operations should be coded so that they are restarted if interrupted.

The interrupt mechanisms employed are platform-specific. Information about each mechanism can be found within the relevant Platform Notes.

5.1.9 Conclusion

Using X•IPC 's asynchronous blocking options, it is possible to have a single program initiate multiple parallel X•IPC operations and to react to them individually as they complete. When used in conjunction with X•IPC 's asynchronous trigger mechanism, it becomes possible to build elaborate event-driven network applications of immense capability–and to do so with relative ease.

5.2 X¨IPC Triggers

X•IPC triggers provide a means for asynchronously monitoring ongoing activity within an instance's QueSys and MemSys subsystems.

5.2.1 QueTrigger() - Defining a QueSys Trigger

A QueSys trigger is a logical link between a QueSys event and a SemSys event semaphore. The semaphore becomes set when the QueSys event occurs.
A QueSys trigger definition has two components:

- The Sid of the event semaphore that is to be set when the monitored QueSys event occurs.
- A specification of the QueSys event that is to be monitored.

The list of QueSys events that can be specified is quite extensive and allows for a wide range of possible trigger specifications. The entire list is given below.
Defining a new QueSys trigger is accomplished using the QueTrigger() function. QueTrigger() takes two arguments:

- The trigger's Sid
- The trigger's QueSys event specification.

Example:

```
/*
 * Create a trigger that will set Sid1 when the number
 * of messages on Qid1 exceeds 75% of its message capacity.
 */

QueTrigger ( Sid1, QUE_T_MSG_HIGH(Qid1, 75) );
```

The above example creates a trigger that will set Sid1 when the message capacity rises above the 75% full mark. This can be used for automatically starting a queue's spooling. A process or thread can wait for Sid1 to be set at which point it could start spooling for Qid1. It may also spawn additional message consuming programs.
It is also possible to create a second trigger that will "fire" when the queue's message capacity dips below 50% full.
Example:

```
/*
 * Set a second trigger to "fire" when the queue drops
 * below 50% message capacity. This trigger uses Sid2.
 */

QueTrigger ( Sid2, QUE_T_MSGS_LOW(Qid1, 50) );
```

This second trigger can similarly be used to automatically turn a queue's spooling off when the queue has emptied below 50%.

The complete list of QueSys event specifications is:

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, N*)	Trigger event when number of bytes written to queue *Qid* 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.

5.2.2 QueUntrigger() - Undefining a QueSys Trigger

A program can undefine a previously defined QueSys trigger by issuing a call to the QueUntrigger() function.
The QueUntrigger() function takes the same pair of arguments as the QueTrigger() function:

- The trigger's Sid

- The trigger's QueSys event specification.

The value of the two arguments must match those that were specified when the trigger was initially defined.
Example:

```
/*
 * Undefine the two triggers defined above.
 */

QueUntrigger ( Sid1, QUE_T_MSG_HIGH(Qid1, 75) );
QueUntrigger ( Sid2, QUE_T_MSG_LOW (Qid1, 50) );
```

5.2.3 MemTrigger() - Defining a MemSys Trigger

A MemSys trigger is a logical link between a MemSys event and a SemSys event semaphore. The semaphore becomes set when the MemSys event occurs.
A MemSys trigger definition has two components:

- The Sid of the event semaphore that is to be set when the monitored MemSys event occurs.
- A specification of the MemSys event that is to be monitored.

The list of MemSys events that can be specified is quite extensive and allows for a wide range of possible trigger specifications. The entire list is given below.
Defining a new MemSys trigger is accomplished using the MemTrigger() function.
MemTrigger() takes two arguments:

- The trigger's Sid
- The trigger's MemSys event specification.

Example:

```
/*
 * Create a Trigger that will set Sid1 when any data is
 * written to the first 1K bytes of segment Mid1.
 */

MemTrigger ( Sid1, MEM_T_WRITE(Mid1, 0, 1024) );
```

The complete list of MemSys event specifications follows:

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.

5.2.4 MemUntrigger() - Undefining a MemSys Trigger

A program can undefine a previously defined MemSys trigger by issuing a call to the MemUntrigger() function.
The MemUntrigger() function takes the same pair of arguments as the MemTrigger() function.

- ☐ The trigger's Sid
- ☐ The trigger's MemSys event specification.

The value of the two arguments must match those that were specified when the trigger was initially defined.
Example:

```
/*
 * Undefine the trigger defined above.
 */

MemUntrigger ( Sid1, MEM_T_WRITE(Mid1, 0, 1024) );
```

5.2.5 Trigger Performance Considerations

There is certain overhead incurred when using X•IPC triggers within an application. This penalty is proportional to the number of trigger definitions in the system.

The correct approach for minimizing overhead is to undefine triggers when they are no longer necessary. Dormant triggers (i.e., triggers that are monitoring events that will never happen) should not be left defined as they can accumulate to slow an application's performance.

5.3 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.

5.3.1 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 *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.

5.3.2 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, the program defines a list of message queues. 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 follow.

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 greater than or equal to 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 less than or equal to n.
QUE_M_PRRNG(Q, n, m)	The first message on queue Q with a priority in the range [n, m].
QUE_M_SEQEQ(Q, seqn)	The first message on queue Q with a value equal to sequence number *seqn*.
QUE_M_SEQGE (Q, seqn)	The first message on queue Q with a value greater than or equal to sequence number *seqn*.
QUE_M_SEQLE(Q, seqn)	The first message on queue Q with a value less than or equal to sequence number *seqn*.
QUE_M_SEQGT(Q, seqn)	The first message on queue Q with a value greater than sequence number *seqn*.
QUE_M_SEQLT(Q, seqn)	The first message on queue Q with a value less than sequence number *seqn*.

The possible *QueSelectCode*s that can be used for selecting a candidate message from one of the listed queues during retrieval operations are 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.

5.3.3 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 in 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.

5.3.4 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.

5.3.5 Conclusion

This presentation 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.

5.4 Understanding QueSys Message Sequence Numbers

QueSys messages are assigned sequence numbers when they are ***sent*** onto a queue. These numbers serve two purposes:

1. To allow an application to compare the relative times that messages were ***sent***, and
2. To allow an application to check that is has not missed any messages coming through a particular queue.

These two objectives are made possible by *two distinct sequence number values* that are assigned to each message when it is inserted onto a QueSys queue: the *QueSys* Sequence Number and the *Queue* Sequence Number. These two values, their usage and interpretations are now described.

5.4.1 The QueSys Sequence Number

Each message that is sent via a call to QuePut() or QueSend() is assigned a unique positive integer value that stamps the sequence, or relative "time," that the message was sent. The QueSys subsystem assigns a *QueSys Sequence Number* to each QueSys message that is sent within an instance, starting with the value '1' from when the instance is started. The QueSys Sequence Number assigned to a message is accessible at the message-header level via the `MSGHDR.TimeVal` field.

The `MSGHDR.TimeVal` value of a message is guaranteed to be unique within an instance's QueSys from the time the instance was started. It is thus possible to use this value to compare two retrieved messages and determine which message was sent first. Example:

```
/*
 * QueGet() two messages from two queues and determine
 * which one was sent earlier.
 */

QueGet(&Msghdr1, QidList(QidA...), ..., QUE_WAIT);
QueGet(&Msghdr2, QidList(QidB...), ..., QUE_WAIT);

if (MsgHdr1.TimeVal < MsgHdr2.TimeVal)
{
    /*  MsgHdr1 message is older   */
    /* (i.e., it was sent earlier) */
}

else
if (MsgHdr1.TimeVal > MsgHdr2.TimeVal)
{
    /*  MsgHdr2 message is older   */
    /* (i.e., it was sent earlier) */
}

else
if (MsgHdr1.TimeVal == MsgHdr2.TimeVal)
{
    /*  IMPOSSIBLE case  */
    }
```

5.4.2 The Queue Sequence Number

Each message is additionally stamped with a second sequence number that marks the sequential position of the message within the specific queue that it is sent. This is referred to as the message's *Queue Sequence Number*. A separate sequence count is kept for each queue created within QueSys.

QueSys assigns a queue-specific sequence numbers to each message that is sent to a queue, starting with the value '1' from when the queue is created. The Queue Sequence Number that was assigned to a message is accessible at the message-header level in the `MSGHDR.SeqNum` field.

The `MSGHDR.SeqNum` value of messages placed onto a queue is guaranteed to be unique *within that queue* from when the queue was created. This allows an application to check that is has received all messages sent through that queue. This is particularly important when the queue is being used as a multicasting channel that has *multiple* programs reading its messages. (See Section 5.5, "QueSys Message Multicasting," for details on this QueSys usage.)

Example:

```
/*
 * QueGet() two messages from a single queue and check
 * that no messages were received off the queue
 * between the two QueGet() calls.
 */

QueGet(&Msghdr1, QueList(QidA...), ..., QUE_WAIT);
QueGet(&Msghdr2, QueList(QidA...), ..., QUE_WAIT);

if (MsgHdr2.SeqNo == MsgHdr1.SeqNo +1)
{
    /*  No messages missed. */
}

else
{
    /*
     * Yes, messages missed.
     * The number of missed messages is:
     * MsgHdr2.SeqNo – MsgHdr1.SeqNo + 1
     */
}
```

5.4.2.1 Sequence Number - Message Select Codes

The QueSys API provides a means for an application to receive a message based on its Queue Sequence Number value. This is accomplished using one of the sequence number message select codes.

Example:

```
/*
 * Receive the message from queue QidA
 * having a queue sequence number of 2.
 */

QueReceive (QUE_Q_ANY,
            QueList(QUE_M_SEQEQ(QidA,2), QUE_EOL),
            ...
            QUE_WAIT);
```

The complete list of sequence number message select codes is found in Section 5.3.2 of the QueSys/MemSys/SemSys User Guide and Section 5.2.2 of the QueSys/MemSys/SemSys Reference Manual.

5.4.2.2 Sequence Number – Using the `QUE_RETSEQ` Flag

The QueSys message retrieval functions—QueGet() and QueReceive()—allow an application to receive either the retrieved message's priority or its Queue Sequence Number value within the call's `*RetVal` parameter. In truth, this is not very important when using QueGet(), since in that case the entire message header is returned containing both values.

This is important, however, when using QueReceive() since in this case the message header is not returned. By default the `*RetVal` parameter is set with the retrieved message's priority. It is possible to override this default by issuing the QueReceive() call specifying the `QUE_RETSEQ` flag as follows:

Example:

```
/* * Receive a message. Get its queue
 * sequence number returned as RetVal.
 */

XINT    RetVal;

QueReceive(..., &RetVal, ..., QUE_RETSEQ | QUE_WAIT);
```

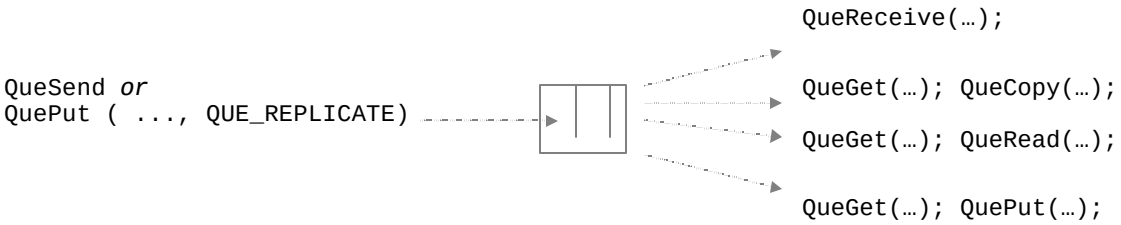
5.5 QueSys Message Multicasting

X•IPC Version 3.0.0 introduces support for two forms of message multicasting over QueSys queues. The two methods address distinct forms of multicast application requirements, described in the following pages.

5.5.1 The "QUE_REPLICATE" Approach

The first form of QueSys message multicasting is the simplest form to code and is geared toward applications where the messages being multicast may occasionally be missed by some of the listeners, depending on the processing speed of received messages. Typical of such applications is that the sent data is constantly being updated, so that the occasional missing of a multicast message is tolerable. Examples include: stock-ticker applications; wind-speed reading applications, etc.

The QUE_REPLICATE option to QuePut() and QueSend() supports this capability. This option, when specified, directs the function to send message copies to multiple (zero or more) users waiting on the specified queue for the sent message. In this case, the messages are never actually placed on the queue. Message copies are sent to those users



that are waiting for the message ***at the time*** of the sender's QueSend() or QuePut() call. This feature has the following general coding approach: The multicaster sends messages using either the QuePut() or QueSend() calls, specifiying the QUE_REPLICATE option as the call's blocking option. Receiving programs can receive messages in several ways, some of which are depicted below.

The *advantage* of the above approach is that:

- It does not require any message selection to be specified by the receiver programs. They simply issue requests for the next message on the queue, and that is what they get. In fact, they get whatever is the next message to be multicast.

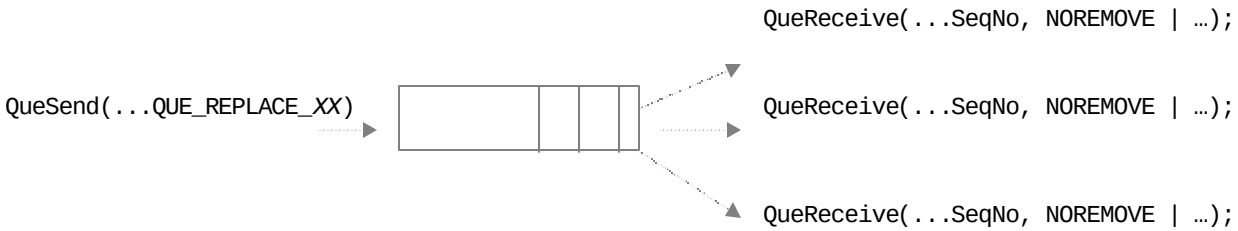
The *disadvantage* of the above approach is that:

- It is possible for receiver programs that are processing messages at a slower rate than the multicaster is sending messages to occasionally miss a multicast message. Unlike the next approach, this method does not provide for slow clients, as no message history is kept.

5.5.2 The "Sliding Queue Window" Approach

This method is a bit more involved than the simple replication approach described above. It employs a number of the QueSys features to create the desired functional effect.

An abbreviated coding description of this approach is as follows.



With this method, the multicaster creates a well-known queue having some significant capacity. This capacity is a key to this approach. Multicast messages are sent to the queue by the multicaster via a QuePut() or QueSend() call. In this case, the sender specifies one of the `QUE_REPLACE_XX` codes as the call's blocking option. This directs *X•IPC* to make room for the message being sent, if necessary, by replacing existing message(s) from one of the queue's end-points. Typically the `QUE_REPLACE_EA` is specified indicating that the oldest (earliest arrived) message is deleted to make room.

Meanwhile, message receivers receive their messages by maintaining a "cursor" within the queue denoting where they are up to. This is accomplished by specifying the `QUE_M_SEQEQ(Qid, SeqNo)` message select code as part of their QueReceive() call, where `SeqNo` is incremented with each message received. This allows a slower receiver to "catch up" to the multicast, without missing any messages. In this case, the queue is in fact being used to maintain the most recent messages multicast.

It should by now be apparent why the capacity of the queue is significant: The larger the queue, the more depth of message history that can be kept for supporting slow receivers.

The advantage of this approach is that:

- It provides for slow clients who can catch missed messages within the limits defined by the message depth of the queue.

The disadvantage of this approach is that:

- It requires more programming work on the receiver side for maintaining some form of "cursor" within the history of multicast messages.

5.6 Using Messages That Have No Text – i.e., Headers Only

Until this point, all QueSys messaging operations that have been described had text-pool data associated with each message. This is usually the case, as messages typically have text-pool data. There are however situations when it is useful to move messages that have no text-pool data associated with them.

X•IPC QueSys supports the ability to move message headers that have no text-pool data. This ability is a performance feature in that its use allows an application to move messages through theQueSys subsystem without needing to access the subsystem's text-pool with each messaging operation.

There are two application situations where this feature is useful:

- Moving messages having small amounts (16 bytes or less) of data
- Moving messages that represent "events"

5.6.1 Small Data Messages

High-performance applications that move messages having 16 bytes or less of data should do so using the user `Data` field within the `MSGHDR` data structure instead of using the text-pool. This is accomplished as follows:

Example:

```
/*
 * Send "Hello World" message header only message
 */

MSGHDR   MsgHdr;

MsgHdr.TextOffset = 0;
MsgHdr.Size = 0;
strcpy(MsgHdr.Data, "Hello World");


QuePut (&MsgHdr, ...);
```

The receiving program retrieves the message using QueGet(), as follows:

Example:

```
/*
 * Receive and print "Hello World" message header
 * only message
 */



MSGHDR   MsgHdr;
QueGet (&MsgHdr, ...);


printf("%s", MsgHdr.Data);
```

Notice that the above examples make no calls to QueWrite() or QueRead(). That is because they are moving header-only messages by means of message text that is inserted directly into the message header (i.e., its `Data` field).

5.6.2 "Event" Messages

A second class of applications that can benefit from this feature are those that use messages as "events" where each message sent represents a discrete application event, where the events are occurring repeatedly, and where each occurrence is significant. Using an event semaphore would not be useful in this case, since it provides no "depth" to record the multiple times that it may have been set. Text-less messages on a queue, however, provide a perfect solution.
Example:

```
/*
 * Send an "event" message.
 */

MSGHDR   MsgHdr;
QuePut (&MsgHdr, ...);
```

The receiving program retrieves the message using QueGet(), as follows:
Example:

```
/*
 * Receive the next "event"
 */

MSGHDR   MsgHdr;
QueGet (&MsgHdr, ...);
```

5.6.3 Programming Semantics

The semantics of QuePut() and QueGet() remain unchanged when working with messages having no text-pool data. All other header-manipulation verbs operate as expected.

5.7 The Queue-Burst Facility for Very High Throughput Message Queuing

Computer applications with high-performance and high throughput requirements are in ever increasing demand as the appetite for computer-generated data increases. General purpose software tools for developing distributed computer applications (i.e., *middleware*) are often not equipped with the functionality needed for building such high-performance applications. As a result, their development cannot benefit from the advantages offered by middleware technology including: API portability, API interoperability, network-transparency and many others.

The X•IPC Queue Burst mechanism is an addition to the general X•IPC API that enables X•IPC users to build high-throughput applications while still benefiting from the high-level programming abstraction provided by X•IPC . The queue-burst mechanism defines a set of function calls that can be used for establishing and sustaining message queuing channels ("*bursts*") between application processes and X•IPC message queues.

As with X•IPC generally, the X•IPC Queue-Burst functions are portable and interoperable on and between all X•IPC -supported environments. Working with the API, the Queue-Burst facility requires virtually no additional network-programming skills.

5.7.1 The Send-Burst

An X•IPC "send-burst" is a mechanism used for supporting application processes that must send messages onto one or more remote X•IPC message queues at a very high rate, as in the following diagram:

An X•IPC user process may start a *send-burst* between itself and queues in a remote instance for carrying out the desired message communication and subsequent enqueuing. Viewed from a more technical perspective, an X•IPC send-burst is implemented as a relationship between a user process and an X•IPC instance, during which time the method of communication between the two is optimized using a specialized high-performance protocol.

A send-burst has a well-defined beginning and end. Burst communication is possible only while the send-burst is active. A function for establishing synchronization points during a send-burst is provided as well.
It is instructive to examine message queuing using the QueSend() verb in order to appreciate the advantages of employing a send-burst. QueSend() semantics define an acknowledgment return code that describes the results (success or error) of the enqueuing operation. A return code is returned *synchronously* to the application - *one for each QueSend() operation*. While this synchronous ACK provides a certain level of reliability, it has the effect of limiting network utilization to one message "in flight" at a time, where the message is followed by an opposite-direction ACK.
A send-burst provides the basic enqueuing capabilities of QueSend() *without* the above performance drawback. Multiple messages are sent in flight over the network, asynchronous to enqueue operation return codes.

An asynchronous mechanism for reporting, for enqueue acknowledgments and for error notifications is provided for tracking message transfer progress.

5.7.1.1 Stand-Alone Functionality

The Queue-Burst API is primarily directed at solving a networking problem. Nonetheless, the definition of the API is portable to stand-alone environments as well. Specifically, it is possible to write and test programs that employ the Queue-Burst functionality within a stand-alone setting by using *X•IPC* 's Stand-Alone or Combined API libraries. The definition of the API does not depend on the presence of a network.
It is worth noting, though, that the notion of *latency* as described at various points within the API definition, is not a practical consideration within a stand-alone setting.

5.7.2 Send-Burst Functions

5.7.2.1 QueBurstSendStart() - Starting A Send-Burst

The parameters to QueBurstSendStart() pre-position message enqueuing parameters at the instance for usage during the subsequent burst enqueuing operations. QueBurstSendStart()'s six arguments are:

- A QueSelectCode to be used for enqueuing messages during the send-burst.

- A QidList to be used for enqueuing message during the send-burst.
- The size of the largest message to be sent in the upcoming burst.
- The size of an internal network read-ahead buffer to be used in the upcoming burst.
- An X•IPC blocking option to be used when enqueuing messages during the upcoming burst.
- An X•IPC asynchronous callback option for handling error reporting during the burst.

A send-burst is started by an X•IPC user is using a QueBurstSendStart() function call. A user that has logged into multiple instances, or that has multiple logins to a single instance, should first connect to the login that will support the burst.
Example:

```
/*
 * Start a send-burst that will enqueue messages onto QidA.
 * Messages will not exceed 64 bytes.
 */


QueBurstSendStart (  QUE_Q_ANY,
                     QueList( QidA, QUE_EOL),
                     64,
                     QUE_BURST_DEFAULT_READAHEADSIZE,
                     QUE_WAIT,
                     QUE_CALLBACK( ErrorFunction, &ErrorAcb)
                   );
```

In the above example, the calling user is defining all enqueuing operations of the upcoming send-burst to use `QUE_Q_ANY` as the QueSelectCode, QueList(QidA, `QUE_EOL`) as the QidList, and `QUE_WAIT` as the blocking option. These parameters are pre-positioned at the instance for use in all QueBurstSend() operations in the burst.
The callback option identifies the user function (i.e., *ErrorFunction*) that is to be called in the event of an error during any of the burst's enqueuing operations. The accompanying ACB (i.e., *ErrorAcb*) passes details of the error to the error function.

5.7.2.2 QueBurstSend() - Send A Burst Message To A Queue

Once a send-burst has been started it is possible to enqueue messages using the burst. The parameters to QueBurstSend() are:

- A possible alternate target Qid.
- A pointer to the message buffer being sent.
- The length, in bytes, of the message being sent.
- The priority to be assigned the message, once enqueued.

QueBurstSend() is a streamlined version of QueSend(). As indicated by its parameter list, only the most basic information regarding the message is required. Details regarding the actual enqueuing operation are not specified. They were prepositioned at the instance via the call to QueBurstSendStart().
Example:

```
/*
 * Send a message for enqueuing. Priority is 1000.
 * Target Qid is selected based on QueSelectCode and
 * QidList specified in QueBurstSendStart() call.
 */

QueBurstSend( QUE_NULL_QID, "Hello World" , 12, 1000);
```

Had the above call to QueBurstSend() followed the earlier call to QueBurstSendStart(), the sent message would be enqueued on message queue QidA, based on the parameters specified in the call to QueBurstSendStart(). It is possible to override the burst's queue selection criteria by specifying a valid Qid as the first parameter to QueBurstSend(). It is thus possible to individually target each sent message, when necessary, as in the following example.
Example:

```
/*
 * Send a message for enqueuing. Priority is 1000.
 * Target Qid, "QidB" overrides the QueSelectCode and
 * QidList criteria specified in QueBurstSendStart() call.
 */

QueBurstSend( QidB, "Hello World" , 12, 1000);
```

QueBurstSend() returns a sequence number (starting with 1), uniquely identifying the sent message, within the current send-burst. The sequence number is used for notifying the user about enqueuing errors. In the event of an enqueuing error, the sequence number of the message involved is asynchronously sent to the user, via the error handling ACB. The number is additionally used for synchronizing a send-burst via QueBurstSendSync(). This is demonstrated in the next section.

5.7.2.3 QueBurstSendSync() - Synchronize a Send-Burst

Enqueuing messages using QueBurstSend() does not provide a per message return code indicating whether sent messages were successfully enqueued. Error reporting, being asynchronous, can suffer from some latency. A situation can arise when an application needs to confirm, in a *synchronous* manner, that all messages sent during the current send-burst have been successfully enqueued. The QueBurstSendSync() operation provides such a synchronization point.
QueBurstSendSync() returns the sequence number of the last send-burst message that was sent *and* successfully enqueued.
QueBurstSendSync() takes as its single argument either the `QUE_WAIT` or the `QUE_CALLBACK` blocking option. See the detailed description of QueBurstSendSync() in the QueSys/MemSys/SemSys Reference Manual for more information.
Example:

```
/*
 * Send 10,000 burst messages. Then confirm that they have
 * all been successfully enqueued.
 */

CHAR *Message = "Sample message";
XINT SeqNo;

for (i=0; i<10000; i++)
{
    QueBurstSend( QUE_NULL_QID, Message, strlen(Message), 2000);
}


SeqNo = QueBurstSendSync(QUE_WAIT);

if (SeqNo != 10000)
{
    /*
     * Messages not enqueued are numbers (SeqNo + 1) to 10,000.
     * One possible remedy is to restart the burst from message
     * SeqNo + 1.
     */

    ...
    ...
}
```

Once synchronized, the send-burst can be restarted by sending additional burst messages.

5.7.2.4 QueBurstSendStop() - Stop A Send Burst

QueBurstSendStop() marks the end of a send-burst. As such, it breaks the relationship between the user process and the instance that supported the burst. It is therefore incorrect to issue any QueBurstSend() calls following the call to QueBurstSendStop() until a new send-burst is started.
Like QueBurstSendSync(), QueBurstSendStop() also returns the sequence number of the last send-burst message that was sent *and* successfully enqueued. The difference is that the latter additionally terminates the burst.
QueBurstSendStop() takes no arguments.
Example:

```
/*
 * Send 20,000 messages.
 */


CHAR *Message = "Another sample message";
XINT SeqNo;

for (i=0; i<20000; i++)
{
    QueBurstSend( QUE_NULL_QID, Message, strlen(Message), 3000);
}

SeqNo = QueBurstSendStop();
```

6. INDEX