Envoy XIPC

Version 3.4.0

XIPC

USER GUIDE

X·IPC VERSION 3.4.0

USER GUIDE

TABLE OF CONTENTS

1. INTRODUCING *X• IPC*

1.1 Purpose

This document presents User guidance for Version 3.4.0 of *X•IPC,* the Extended Interprocess Communication Facilities product from Envoy Technologies Inc.

X•IPC is a toolkit for developing software systems employing Interprocess Communication (IPC). *X•IPC* is comprised of four IPC subsystems, each with a library of functions and support utilities:

☐ MomSys, the Message Oriented Middleware subsystem

The *X•IPC* message oriented middleware subsystem, 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.

☐ QueSys, the Message Queue System
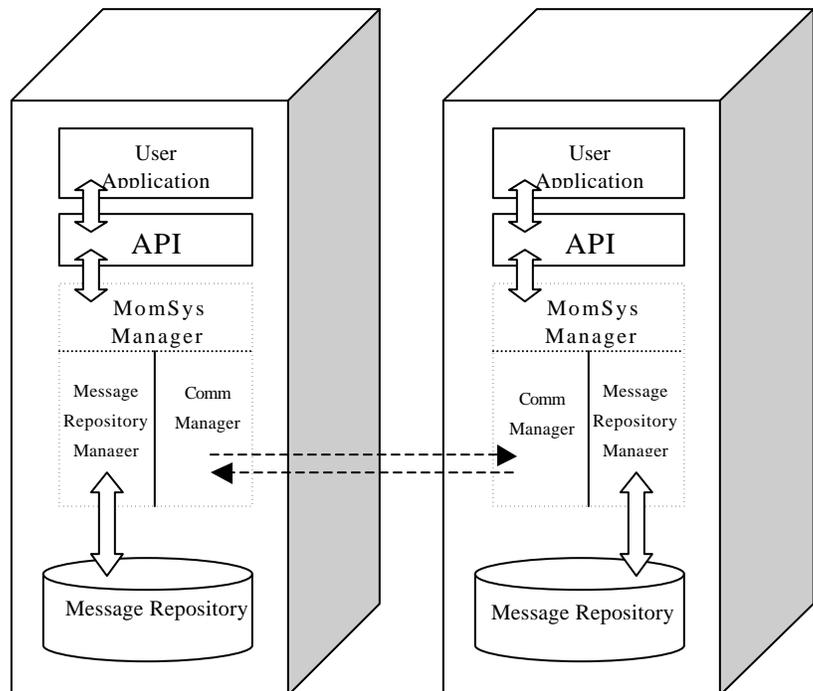
The *X•IPC* message queue system, QueSys is a complete message queuing facility. Many advanced features are included (e.g., individualized queue sizing, dynamic queue spooling, queue multiplexing, etc.) to facilitate most necessary message queuing requirements.

☐ 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.

☐ MemSys, the Shared Memo ry 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.

This User Guide describes the *X•IPC* product without specific reference to the individual subsystems. It provides a global presentation of *X•IPC* concepts–most importantly, *instances*–and usage; it is accompanied by a Reference Manual. Separate documentation is available for the subsystems. Used together or individually, these subsystems provide significant enhancements to the native IPC facilities of the supported operating systems. *X•IPC* additionally provides its full functionality distributed over a network.

X•IPC is a set of libraries and support utilities that greatly simplifies software development involving stand-alone or network IPC. *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.

X•IPC benefits are realized throughout all phases of the software development cycle:

☐ System design is simplified by the availability of *X•IPC* 's rich set of basic and advanced IPC functions. System design decisions are no longer constrained by non-existent or deficient IPC capabilities.

 Distributed application design also benefits from *X•IPC* 's full functionality being transparently available over a network.

Database Servers   DBMS

Application and
Communication
Servers   UNIX, NT, VMS, OS/2

Workstation
Clients   Java, Win95, WinNT, Mac, UNIX, OS/2 …
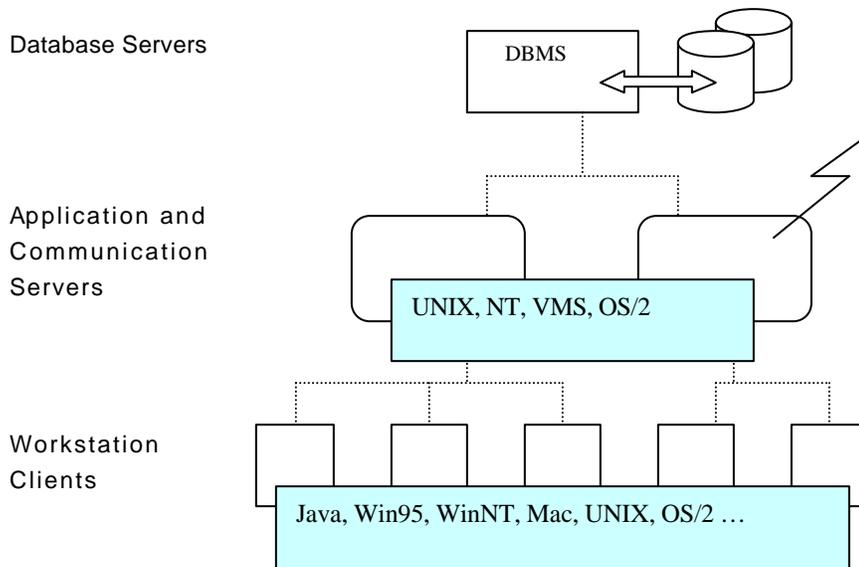
☐ The development phase is enhanced at the unit test level and, more significantly, during system integration. IPC problems resulting from programmer error or misinterpretation are identified and resolved rapidly via *X•IPC* 's on-line monitoring and debugging facilities.

 Network application development is further simplified by the fact that all IPC-related development can occur in a stand-alone environment and subsequently be distributed over a network with virtually no modification.

☐ System maintenance is similarly enhanced by the ability to remotely monitor live (stand-alone or network) applications in the field, if and when they exhibit problems.

☐ Porting IPC-laden systems between dissimilar operating systems is reduced to a recompile instead of a total redesign.

☐ Spreading an application over a network requires no program modification. Distributed processes can communicate using *X•IPC* functionality regardless of their network location.

X•IPC provides network-transparent connectivity

In short, *X•IPC* redefines how the IPC components of sophisticated multitasking, multi-platform and distributed software systems are designed and developed.

1.2 Scope

This *X•IPC* User Guide is for experienced software developers, who are familiar with the basic concepts of IPC as well as with common software development practices. These developers need the enhancements provided by *X•IPC* for easing and expediting the development of quality, portable applications in a multitasking or distributed environment.

This volume, the *X•IPC* User Guide, is put to best advantage if you first read Chapters 1 through 4 thoroughly to become well acquainted with the *X•IPC* product and its key concepts. Chapter 5 addresses basic programming techniques, while Chapter 6 presents more advanced topics in greater detail. Technical Notes, which discuss special issues and product enhancements, are provided in Chapter 7, the Appendix.

1.3 Availability

X•IPC is available on a wide variety of operating system platforms and, when used in a networking environment, includes support for a wide range of protocol families. This platform and environmental independence is one of the major benefits of working with *X•IPC*: It provides for immediate IPC source code portability. It additionally allows for flexible configuration of a distributed application's processing components, since they are not bound to any particular operating system platform.

Note that all platform-specific information, from installation and program compiling/linking to configuration and administration guidance, is found in the individual Platform Notes documentation that is available for each supported platform.

1.4 Documentation Roadmap

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

☐   Getting Started is a brief introduction to *X•IPC*. It 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 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 provides subsystem-level 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 provides subsystem-level functions and macros, interactive commands and sample programs, as well function calling sequences and possible return codes.

1.5 Getting Started

1.5.1 SYSTEM REQUIREMENTS

X•IPC utilizes certain native operating system resources when it is used. The quantities and sizes of these resources are relatively small and are usually available within the standard operating system configuration. Formulae for calculating required native resources are operating-system dependent and are described in the Platform Notes accompanying the product.

When using *X•IPC* in a networking environment, certain network resources are used. The nature and quantities of these resources are network-dependent and are outlined as well in the Platform Notes.

1.5.2 INSTALLATION

Installation is operating system and network dependent. Consult the Platform Notes for details of the installation procedure.

2. *X•IPC* CONCEPTS

2.1 Interprocess Communication (IPC)

2.1.1 MULTITASKING – STAND-ALONE IPC

With the emergence of powerful microprocessors, multitasking operating environments have become increasingly popular, most recently at the micro-computer level. This is a direct result of the increased processing power provided by these processors. Such power is a prerequisite for an operating system performing as a multitasking scheduler.

The popularity of UNIX systems, from workstations to super-micros, the increasing acceptance of Windows NT and Windows 95, along with the continued popularity of OS/2 and VMS, are all indicative of the movement toward employing sophisticated multitasking programming techniques for solving increasingly complex system requirements.

The key to such systems is Interprocess Communication. IPC is the mechanism by which multiple active tasks dynamically synchronize and pass information between one another. IPC provides the tools for affecting process synchronization, message passing and resource and memory sharing needed within the context of multitasking systems.

2.1.2 DISTRIBUTED COMPUTING - NETWORK IPC

More recently, a second form of IPC requirement has grown in demand. While processing power has become less expensive and increasingly diversified, network technologies have matured and become widespread. The convergence of these factors has led to an upsurge in demand for distributed computing applications. Of particular interest is the growing need for guaranteed message delivery.

Such distributed applications often have the same kinds of IPC requirements as stand-alone multitasking IPC applications: interprocess synchronization, message passing and resource and memory sharing. The difference is that the processes in a distributed application are not confined to one computing platform and may be spread across a network.

2.1.3 GUARANTEED MESSAGE DELIVERY

IPC tools have facilitated the successful design and implementation of multitasking and distributed application systems which can make possible the building of reliable, large-scale mission critical distributed applications which demand guaranteed message delivery. The spread of these enterprise-wide applications has been accompanied by a demand for guaranteed message delivery. As the technological environment increases in complexity, incorporating disparate operating systems, platforms and applications, the risks associated with messaging among them has risen dramatically. Only with guaranteed message delivery provided by IPC network tools can such environments by operated and expanded with reliability, with the confidence that *messages cannot be lost*.

2.2 Why *X•IPC*?

A major shortcoming of earlier IPC approaches is that they reflect the state and limitations of software development methods of a decade ago. This is most apparent in the areas of IPC software engineering techniques, IPC functionality, IPC source code portability, network IPC transparency and system scalability.

X•IPC overcomes these problems in a consistent and cohesive manner.

2.2.1 *X•IPC'S* ADVANCED IPC SOFTWARE ENGINEERING TOOLS AND METHODS

X•IPC provides the systems developer with a set of IPC tools and techniques that support the latest programming methods. For instance:

☐ On-line monitoring and interactive debugging at the IPC level of a system's execution is not possible with current IPC tools. *X•IPC* provides the developer with the ability to view, in real-time, all of his IPC resources as they are created, manipulated and deleted–irrespective of whether the IPC activity is occurring on a stand-alone platform or dispersed over a network.

This facility reduces time spent in testing and integration phases of a system's development. In addition, it enhances product support efforts by providing the ability to remotely monitor live production systems in the field if and when they exhibit problems.

☐ *X•IPC* makes it possible to develop a distributed application using a stand-alone IPC environment, and to subsequently disperse the application's component processes to positions on a network with virtually no IPC code modifications.

☐ *X•IPC* moves all aspects of IPC configuration and parameterization out of the kernel. Most current IPC tools place any possible IPC parameterization together with other kernel-related parameter values. This, in effect, forces all applications to share in the parameterization decisions and in the resulting IPC resource allocation pools, regardless of differing and sometimes conflicting needs. And when changes are agreed upon, they can only be affected after the kernel has been brought down, thus interrupting everyone using the system.

The setting and tuning of IPC parameter values is done at the application level, each application according to its own specific requirements. Concurrently active *X•IPC* intensive applications have no relationship with one another and can be configured and fine-tuned individually. *X•IPC* monitoring and debugging is performed and segregated on an application-dependent basis as well.

2.2.2 *X•IPC'S* ENHANCED IPC BASIC AND EXTENDED FUNCTIONALITY

The native IPC facilities of the various operating systems are frequently inadequate. *X•IPC* affords the developer a more complete set of IPC functionality. New basic and advanced IPC features are provided. A few examples follow.

♦ *X•IPC* provides guaranteed message delivery––assuring that *no message can be lost*–– to support the demands of today's large-scale, mission-critical, globally distributed applications.

♦ The queue system provides individualized queue sizing in terms of bytes and/or messages, thus allowing for throttling of message-producing tasks. Automatic spooling for overflowing queues is also provided as an option in order to avoid losing peak-period messages.

♦ Atomic operations involving multiple queues provide the multiplexing functionality often needed for building complex systems such as transaction processing monitors. Messages designated "oldest," "youngest," "highest-priority," etc., can be retrieved atomically from groups of queues.

♦ *X•IPC's* fully functional queue system eliminates the shortcomings of some of the underlying native IPC facilities. For example, queue ownership restrictions inherent in OS/2 are removed.

♦ Additional *X•IPC* functions include a comprehensive implementation of event and resource semaphores. Multiple semaphores can be operated on in single *X•IPC* operations thus allowing for a variety of waiting and acquiring alternatives ("any," "all-atomic," "all-cumulative").

♦ *X•IPC's* shared memory system provides for memory read-and-write locking and protection at the byte level. This too is unique to *X•IPC*.

X•IPC additionally supports synchronous and asynchronous operations. *X•IPC* also supports asynchronous triggers that monitor specific aspects of an application's IPC environment (e.g., queue "xyz" rises above 90% capacity).

Many more enhancements exist and are described in their appropriate sections below.

The wide array of additional *X•IPC* functional capabilities elevates the task of system design to a higher and more abstract level. The difficult job of reducing complex system requirements to meet the low-level realities of native IPC functionality is significantly alleviated.

2.2.3 *X•IPC'S* IMMEDIATE INTER-OPERATING SYSTEM IPC SOFTWARE PORTABILITY

The API used to access *X•IPC* is independent of the host operating system. Thus, the IPC components of a system written using *X•IPC* are immediately portable from one operating system to another.

The most difficult part of porting an application between operating systems is often the IPC portion. This is due to the gross dissimilarities in functionality, calling sequences and underlying IPC methodologies employed by the operating systems involved. Bridging these differences frequently requires extensive modifications to the code and sometimes a total redesign. In such cases, multiple versions of source code have to be maintained and kept in sync.

In contrast, portable IPC code is an immediate by-product of using *X•IPC*. . The benefits are manifold:

☐ There is no need to maintain multiple versions of a multi-platform application's source code, or to edit the source code for porting. The cost of version control is significantly reduced.

☐ System architects can design multiple-platform applications based on the application's requirements, rather than according to the lowest common denominator IPC constraints of the specific platforms involved.

☐ In-house expertise of the native IPC facilities for each operating system is no longer necessary. Training new programmers in IPC coding is performed once, regardless of the operating system to be used.

2.2.4 *X•IPC'S* NETWORK IPC TRANSPARENCY

X•IPC presents a uniform approach for handling both stand-alone and distributed forms of IPC. Processes synchronize, communicate and share data with one another using the *same* set of function calls whether they are on a single multitasking platform or distributed over a network of heterogeneous platforms .

The resulting benefits for the developer are:

☐ Full *X•IPC* functionality is extended transparently across a network.

☐ The need for network programming expertise is eliminated.

☐ Operating system differences are no longer an IPC concern.

☐ Network protocols are no longer an IPC concern.

☐ Stand-alone multitasking IPC applications can be distributed over a network with virtually no code modifications.

3. THE *X•IPC* PLATFORM

Before a computer platform can be used for supporting *X•IPC* activity, the appropriate underlying *X•IPC* environment must first be established on that platform. This environment is referred to as the "*X•IPC* Platform Environment." This section discusses the following aspects of the *X•IPC* Platform Environment:

- Function of the *X•IPC* Platform Environment
- *X•IPC* Platform Environment Configuration
- *X•IPC* Platform Environment Commands

3.1 Function of the *X•IPC* Platform Environment

As stated above, a computer platform that is to support any form of *X•IPC* activity must first have its *X•IPC* platform environment started. A platform's *X•IPC* environment encompasses a number of background processes as well as underlying system data structures.

The *X•IPC* platform environment is the infrastructure used to support all *X•IPC* activity on that platform. Components within the *X•IPC* platform environment include:

- an internal *X•IPC* instance that is used by *X•IPC* for supporting internal interprocess communication within the platform

- a number of *X•IPC* daemon/service programs that operate in the background for supporting various *X•IPC*-related functions, such as: catalog and namespace services, asynchronous operation services, idle-user detection services, etc.

3.2 *X•IPC* Platform Configuration

The *X•IPC* platform environment must be properly configured in order for *X•IPC*-based applications running on the platform to operate properly. This configuration is based on a single configuration file, called `xipc.env`. The `xipc.env` file is read by an *X•IPC* utility command, `xipcinit,` to start the *X•IPC* platform envuironment. The `xipcinit` command reads the parameter definitions contained within the `xipc.env` file for setting up the *X•IPC* platform environment. (The location and contents of the `xipc.env` file will be discussed below.)

The `xipc.env` file supports a set of parameter definitions that describe the nature of the *X•IPC* platform environment that will be started. Some of these parameters will be described in the following sections. The complete list of parameter sections, parameter names and possible parameter values are listed on the <u>*X•IPC*</u> <u>Reference</u> <u>Manual</u> page for the `xipcinit` command.

3.2.1 *X•IPC* PLATFORM CLASSIFICATION

There are two general forms of *X•IPC* platform configurations: "*X•IPC* Server Platform Configuration" and "*X•IPC* Client Platform Configuration". These are now described.

3.2.1.1 *X•IPC* Server Platform Configuration

A computer platform which will be used for supporting one or more *X•IPC* instances is referred to as an "*X•IPC* Server Platform."

The two typical situations where server platform configuration is required are as follows:

- In configuring an *X•IPC* platform that will host an *X•IPC* instance that is to be accessed by local and/or remote QueSys / MemSys / SemSys users, as depicted in the following diagram:

Local
User
Progra

Remote
User
Progra
...

Q
S
M

X•IPC Instance

Remote
User
Progra
...
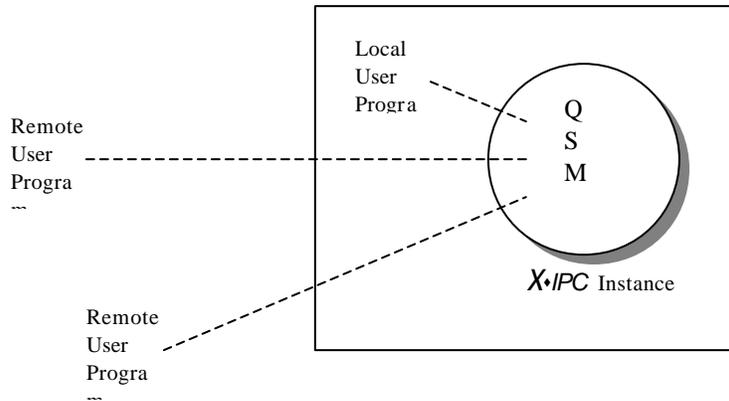
- In configuring an *X•IPC* platform that will be used as part of a MomSys application, as depicted in the following diagram:

MomSys
User
Program

X•IPC Instance

MomSys
User
Program
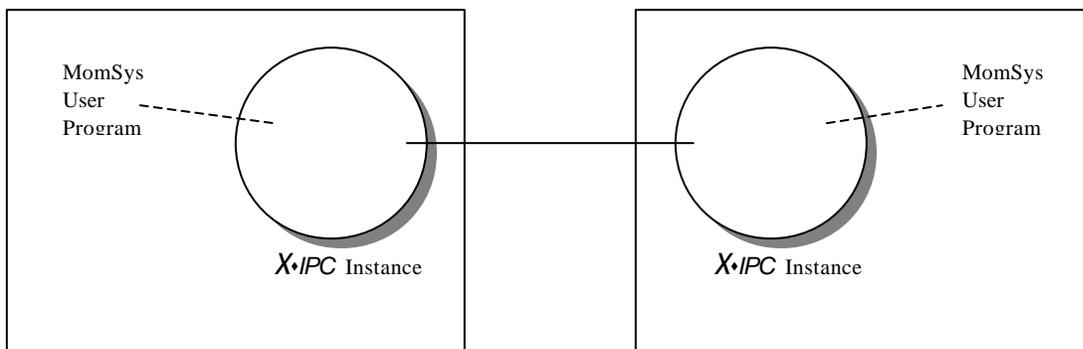
X•IPC Instance

Note that programs employing the MomSys subsystem require a local *X•IPC* instance to be active for supporting the MomSys activity. Hence, MomSys users must configure all involved computer platforms as *X•IPC* server platforms.

The `xipc.env` file for such a configuration requires that all *X•IPC* services/daemons be started on that platform. This is the default behavior for an `xipc.env` file that does <u>not</u> specify the `START` parameter within the file's [`xipcinit`] section. A default `xipc.env` file for an *X•IPC* server configuration has the following contents:

Default *X•IPC* Server Platform Configuration (`xipc.env`)
...

```
[xipcinit]

[catalog]

[catalog.tcpip]
```

3.2.1.2 X•IPC Client Platform Configuration

A computer platform that will <u>not</u> support any *X•IPC* instances is referred to as an "*X•IPC* Client Platform." Thus, for example, a platform that is to be used for supporting QueSys / MemSys / SemSys programs that access <u>remote</u> *X•IPC* instances exclusively may be configured as an *X•IPC* client platform.
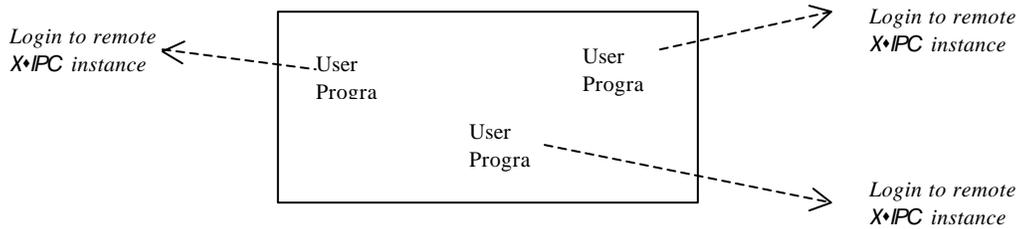
The `xipc.env` file for such a configuration requires that a limited subset of *X•IPC* daemon/service programs be started on that platform. This is specified via the START parameter within the `xipc.env` file's `[xipcinit]` section. An example of such a file is as follows:

Typical *X•IPC* Client Platform Configuration (`xipc.env`) File

```
[xipcinit]
START        xipciad   # Only starts this program.

[catalog]

[catalog.tcpip]
```

Where the platform will not employ any of *X•IPC*'s asynchronous functionality, the `xipciad` name can be deleted from the START parameter. In such a case, the START parameter should appear with no values assigned to it, as follows:

X•IPC Client Platform Configuration (`xipc.env`) File supporting no asynchronous functionality or catalog access.

```
[xipcinit]
START    # Has no parameters. Starts no background programs.

[catalog]

[catalog.tcpip]
```

3.3 *X•IPC* Platform Commands

The two *X•IPC* commands that initialize and terminate a platform's *X•IPC* platform environment are `xipcinit` and `xipcterm`, respectively. These commands make particular use of the `XIPCROOT` environment variable that must be defined in order for them to function.

3.3.1 THE `XIPCROOT` ENVIRONMENT VARIABLE

When *X•IPC* is started on a platform--via the `xipcinit` command--*X•IPC* sets up its internal platform environment for supporting all subsequent *X•IPC* activity on that platform. As part of this initialization, `xipcinit` reads the `xipc.env` platform configuration file to ascertain which platform-wide resources need to be set up. The location of the `xipc.env` file is defined by the `XIPCROOT` environment variable. If `XIPCROOT` is not set, or if it is set, but points to a directory/folder having no `xipc.env` file, the `xipcinit` command will fail.

```
                                            XIPCROOT

            X•PC Installation
            Directory/Folder

            …xipc.env
            …scratch-pad
            files
            …bin/
            …include/
            …lib/
            …samples/
            …log/
```
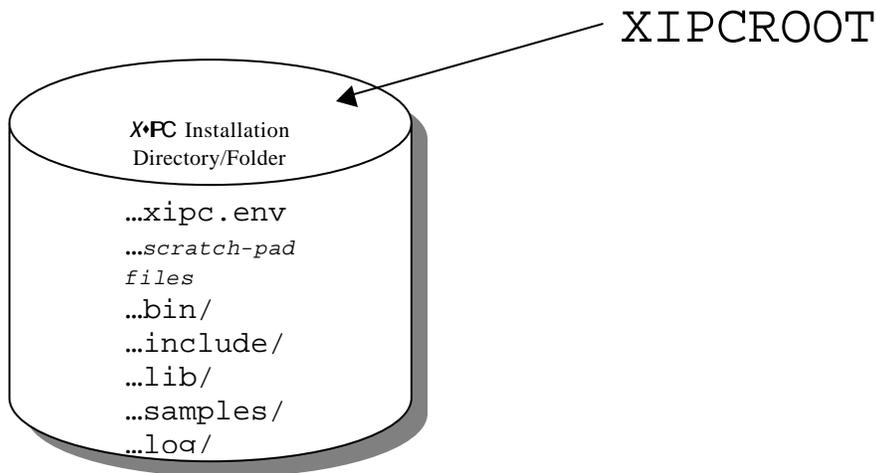
`xipcinit`'s work includes the creation of a number of scratch-pad files within the `XIPCROOT` directory. As such, the `XIPCROOT` directory must be situated within a <u>writeable</u> area of the file-system.

The `XIPCROOT` environment variable typically has a second function: defining for `xipcinit` where to find the installed *X•IPC* product. Thus, `xipcinit` uses the value of `XIPCROOT` to find the product's `bin` directory for starting internal processes that are installed therein.

Typically, these two roles of `XIPCROOT`:

♦   identifying the directory in which `xipcinit` will find the `xipc.env` file and in which it will create scratch- pad files, and

♦   identifying the location of the installed *X•IPC* product

are addressed in a unified manner. In such a case, `XIPCROOT` is set pointing to the installation directory in which the `xipc.env` file is positioned and in which `xipcinit` creates scratch-pad files. This is depicted in the above diagram.

Occasionally, it is desired to have the *X•IPC* product installed in an area of the file system that is read-only. In this case, it is undesirable (and, in fact, impossible) for xipcinit to use the installation directory for its scratch-pad files.

The prescribed approach, therefore, is to move the xipc.env file and the log directory to a writeable directory and set XIPCROOT pointing there. xipcinit now knows where to find xipc.env and where to do scratch-pad file work, when invoked. xipcinit must still, however, be guided to the *X•IPC* installation directory, which is elsewhere (in a read-only directory); this is accomplished by adding, within the [xipcinit] section of the xipc.env file, the XIPCPATH parameter set with the directory path of the installed *X•IPC* product. This is depicted in the following diagram:

XIPCROOT


```
[xipcinit]
XIPCPATH    # set this parameter to point to the
            # X•IPC  installation directory/folder

[catalog]

[catalog.tcpip]
```
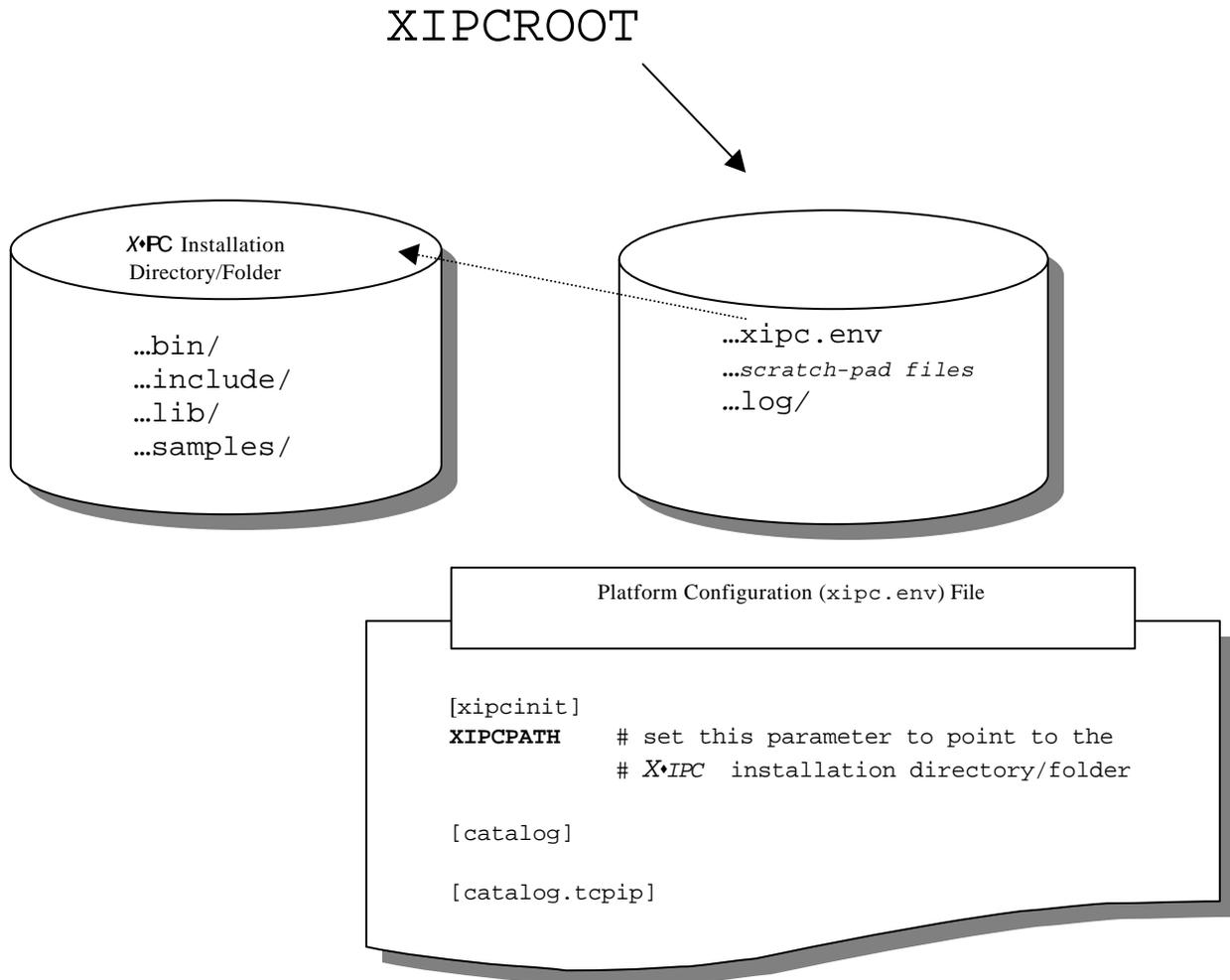
3.3.2 THE xipcinit COMMAND

The xipcinit command is used for initializing the *X•IPC* platform environment. xipcinit <u>must</u> be the first *X•IPC* command issued on the platform when the platform is started. xipcinit reads the xipc.env file and sets up all internal structures and background processes needed for supporting *X•IPC* activity on the platform, based on the file's parameter settings.

xipcinit requires the setting of the XIPCROOT environment variable as described above. It takes no arguments. See the *X•IPC* <u>Reference</u> <u>Manual</u> for parameter guidelines.

3.3.3 THE `xipcterm` COMMAND

A bracketing command, `xipcterm`, is used to terminate the *X•IPC* environment on a given platform. `xipcterm` should be the last *X•IPC* command issued when the platform is stopped. `xipcterm` closes all internal structures and background processes needed for supporting *X•IPC* activity on the platform.

`xipcterm` takes no arguments, but it too requires the `XIPCROOT` environment variable to be set.

Refer to the *X•IPC* Reference Manual for details on `xipcinit, xipcterm` and for information about the parameters that may be defined within the `xipc.env` file. The `XIPCROOT` environment variable is defined as well in the same Reference Manual.

3.4 *X•IPC* Logging

There are two types of logging relevant to *X•IPC* : *Platform Environment Logging* and *Instance Logging*. Descriptions follow.

3.4.1 PLATFORM ENVIRONMENT LOGGING

X•IPC 's platform environment generates a set of log files containing information about the activities occurring within the *X•IPC* platform environment. These log files are generated within the *X•IPC* platform's log directory (i.e., the `log` directory/folder within the directory/folder pointed at by the `XIPCROOT` environment variable. Refer to the diagrams earlier in this section.)

These log files are actually divided into two groups:

- The single *X•IPC* system summary log file - `xipcsys.log`

- Individual log files for each of the platform-level background service/daemon programs

3.4.1.1 *X•IPC* System Summary Log File – `xipcsys.log`

The `xipcsys.log` file provides a high-level running summary of *X•IPC* activity occurring on a platform. Included within this file are entries such as:

- "`xipcinit` has initialized the platform environment"
- "`xipcterm` has terminated the platform environment"
- "User xipc instance … was started"
- "User xipc instance … was stopped"

The `xipcsys.log` file will also include reports of high-level warnings or errors occurring within other *X•IPC* components. As such, the `xipcsys.log` file is the central repository of overall *X•IPC* activity occurring on a platform; it should be examined first when tracking down suspected problems.

3.4.1.2 Background Service/Daemon Logging

Each background service/daemon program may log errors and warnings specific to its function within a log file specific to it. As with the `xipcsys.log` file, these log files are generated within the platform's `log` directory. These files follow the naming convention `<ProgramName>.log`. For example, the `xipcisd` background program generates the `xipcisd.log` file.

3.4.2 INSTANCE LOGGING

A second form of logging occurs at the *X•IPC* instance level and, then, only within instances running the MomSys subsystem. In this case, a series of log files, specific to the MomSys subsystem of that instance, are generated in the instance's anchor directory/folder (i.e., the directory/folder in which the instance's configuration file is situated).

Assuming that the instance's configuration file was named `test.cfg`, then the generated MomSys log files will have names such as: `test.SSS`, `test.MRI`, `test.MRO`, `test.CSI`, `test.CSO` and `test.CLK`, corresponding to the internal components of the instance's MomSys subsystem.

4. X•IPC INSTANCES

4.1 What is an *X•IPC* Instance?

An important *X•IPC* concept is that of an *instance*. An *X•IPC* instance is an environment for doing *X•IPC* work. An *X•IPC* instance is comprised of one or more of *X•IPC*'s subsystems: MomSys, QueSys, SemSys and MemSys.

In the case of QueSys, SemSys and MemSys, an *X•IPC* instance is the true *hub* of *X•IPC* activity; it is in this context that instances are discussed here.

In the case of MomSys, instances serve as *gateways* to route messaging activity; MomSys' utilization of instances is discussed at greater length in the MomSys documentation.

In general, the reader should refer to the respective subsystem manuals for further information on the establishment and use of instances and for all other subsystem-specific details.

X•IPC instances are typically utilized on an application-by-application basis, with instances defined to meet the specific IPC requirements of a given application and configured to optimize IPC performance of the application.

In short, an *X•IPC* instance is an IPC environment tailored to the specific needs of an application and its programs that will use it.

4.2 Defining an *X•IPC* Instance

An *X•IPC* instance is defined via an *X•IPC* instance configuration file. An instance configuration file is a flat ASCII file containing the parameterization definitions that describe the *X•IPC* instance. *X•IPC* instance configuration file names have the form:

```
<InstanceFileName>.cfg
```

Examples for various platforms:

```
frontend.cfg            stress.cfg            userdisk:[email]production.cfg


/usr/demo.cfg                 tpsys.cfg


/tmp/test.cfg                 c:\appls\transact.cfg
```

The maximum length of *InstanceFileName* depends on the host operating system's file naming rules.

4.3 Configuration (.cfg) Files

An *X•IPC* configuration file (.cfg file) completely describes an *X•IPC* instance. As such, it contains all necessary information needed to describe and parameterize the instance.

The configuration file is comprised of separate sections for each of the *X•IPC* subsystems. The sections contain the definitions and parameters specific to that subsystem. An additional section is required for instances that will be used in a network environment.

The order of the sections is insignificant. Blank lines and comments are ignored. Comment lines can be started using almost any non-alphanumeric character. Comments can also occur in lines following any parameter definition. The exact details of the syntax are specified in the *X•IPC* Reference Manual, as well as in the individual subsystem manuals.

The sample configuration file below defines an *X•IPC* instance for an E-Mail application having an Instance File Name of /usr/email. It references the QueSys, MemSys and SemSys subsystems, whose documentation should be referred to for further information.

A transaction processing application would likely define its own IPC environment separately, in a different configuration file, perhaps named tpsys.cfg. Certain supported operating systems require additional operating system specific parameters to be specified within their configuration files. These parameters (if any) are listed and described in the Platform Notes for the respective platforms.

```
#==============================================================
#
# File: /usr/email.cfg
# Created: May 22, 2001
#
#--------------------------------------------------------------
#
# This XIPC instance is used for demonstrating the basic
# functions of our E-Mail application. Various limitations are
# imposed on system capacity since it is a demo.
#
#--------------------------------------------------------------

[SEMSYS]
MAX_SEMS     10       # the demo system uses 10 semaphores.
MAX_USERS        10       # five programs plus expected async activity.
MAX_NODES        50       # general formula given in Reference Manual.

[QUESYS]
MAX_QUEUES       20       # never needs more than 20 queues.
MAX_USERS        10       # programs plus expected async activity.
MAX_NODES        50
MAX_HEADERS      100      # worst case: assuming heavy traffic.
SIZE_MSGPOOL 48       # worst case: assumes download activity.
SIZE_MSGTICK 64       # smallest message is 64 bytes.
SIZE_SPLTICK 128

[MEMSYS]
MAX_SEGS     15       # depends on length of demo.
MAX_USERS        10       # programs plus expected async activity.
MAX_NODES        50
MAX_SECTIONS 100
SIZE_MEMPOOL 32       # must not be less than 16 K for demo.
SIZE_MEMTICK 256      # smallest segment to be used.

[REMOTE_USER]
MAX_TEXTSIZE 1024

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

4.4 Defining An Instance Having A Null Subsystem

In some situations it may be desirable to define an *X•IPC* instance that supports a subset of *X•IPC*'s subsystems. For example, one applications may require an instance that only uses QueSys message queuing, while a second application may have the need for an instance that supports semaphores and shared memory.

For addressing these situations, the developer can define an instance that has a subset of its subsystems designated as null. Such an instance will support only those IPC services corresponding to the subsystems that are defined. Attempts to issue *X•IPC* operations using the services of the null-defined subsystems are returned as an error.

```
#===============================================================
#
# File: /usr/subset1.cfg
# Created: May 22, 2001
#
#---------------------------------------------------------------
#
# This XIPC instance demonstrates configuring an instance that
# has no QueSys (or MomSys) subsystems.
#
#---------------------------------------------------------------

[SEMSYS]
MAX_SEMS     10
MAX_USERS        10
MAX_NODES        50

[MEMSYS]
MAX_SEGS     15
MAX_USERS        10
MAX_NODES        50
MAX_SECTIONS 100
SIZE_MEMPOOL 32
SIZE_MEMTICK 256

[REMOTE_USER]
MAX_TEXTSIZE 1024
#===============================================================
```

The above instance is defined to have a null QueSys (and a null MomSys), simply by virtue of omitting them as section headers.

It is important to bear in mind that if a subsystem is defined as null within an instance, then no *X•IPC* operations of that nature are possible within that instance. As an example, in the above defined instance, it would be an error to issue a QueSys or MomSys command.

The advantage of using a null subsystem is that doing so reduces the memory size of an instance (i.e., the amount of native shared memory required by *X•IPC* for supporting the instance).

4.5 XIPCROOT

`XIPCROOT` is the platform directory environment variable. It is required in all cases, for stand-alone, local and network instances alike.

When *X•IPC* is started on a platform--via the `xipcinit` command--*X•IPC* sets up its internal platform environment for supporting all subsequent *X•IPC* activity on that platform. As part of this initialization, `xipcinit` reads the `xipc.env` platform configuration file to ascertain which platform-wide resources need to be set up. The location of the `xipc.env` file is defined by the `XIPCROOT` environment variable. If `XIPCROOT` is not set, or if it is set, but points to a directory/folder having no `xipc.env` file, the `xipcinit` command will fail. See chapter 3 for a more detailed description.

4.6 Starting an *X•IPC* Instance

An *X•IPC* instance must be started before it can be used. (And, recall that the *X•IPC* platform configuration environment must be initialized using `xipcinit` before *any* *X•IPC* activity can be initiated.). This is accomplished using the `xipcstart` command. `xipcstart` starts an *X•IPC* instance. Its argument is the instance configuration file name of the instance to be started––or to be more precise, the full or relative path name of the instance configuration file *excluding* the `.cfg` suffix.

Consider the following UNIX example:

```
    xipcstart /projects/local/tpsys
```

Programs attempting to use an instance that has not been started will receive an error code indicating the problem. This will be elaborated on in the discussion of "login" functions below.

The Instance File Name can be omitted from the command line. In such a case, `xipcstart` uses the value of the `XIPC` environment variable as the Instance File Name of the instance to start.

Starting an instance that is to be used in a network environment requires an additional command argument. This is described in the *X•IPC* Reference Manual.

As described in the Advanced Topics section of this User Guide, an instance can also be started under program control.

4.6.1 TEST STARTING AN INSTANCE

`xipcstart`, when executed, generates a report that specifies the amount of native operating system memory resources required by the instance.

It is possible to have `xipcstart` run in test mode, so that it produces a report indicating the memory resources that *would* be required by the instance, had the instance actually been started, and to *not* actually start the instance. This mechanism is useful for scoping the size of an instance before it is actually started. The test flag (`-t`) directs `xipcstart` to produce a test report regarding an instance.

Examples:

```
    xipcstart /projects/local/tpsys -t
```

4.7 Stopping an *X•IPC* Instance

An *X•IPC* instance is stopped using the `xipcstop` command. `xipcstop` terminates an active instance, and releases all resources associated with that instance. Its argument is the Instance File Name of the instance to be stopped.

Example:

```
xipcstop /projects/local/tpsys
```

The above command stops the *X•IPC* instance that had been started using the Instance File Name shown (`tpsys`). Programs requiring the instance's *X•IPC* facilities can no longer be run.

Programs still using an instance at the time that it is stopped receive an error code indicating the stoppage of the instance.

Here, too, the Instance File Name can be omitted from the command line. In such a case, `xipcstop` also uses the value of the `XIPC` environment variable as the Instance File Name of the instance to stop.

As described in the Advanced Topics section of this book, an instance can also be stopped under program control.

4.8 User-Controlled Configuration

As we have seen, each *X•IPC* instance is individually configured by the user, without the need to modify the operating system kernel parameters.

This has a number of obvious advantages:

☐ Each application's *X•IPC* environment can be configured and optimized according to its own specific IPC needs.

☐ *X•IPC* configurationchanges can be applied without affecting the *X•IPC* instances of other applications.

☐ Special *X•IPC* instances can be devised for testing various aspects of an application's performance. Such test instances can be used to verify the correctness of special case logic within a system by artificially forcing those special situations to "occur." Examples include borderline testing (e.g., insufficient message headers: create an instance with an artificially low number of headers), and stress testing (e.g., insufficient shared memory space: create an instance having an artificially small-sized MemSys). In this manner, obscure paths in a system's code can be thoroughly tested.

☐ Production copies of a system can be individually tailored for different customer and/or site requirements, without the need to modify the kernel at each site.

4.9 Multiple *X• IPC* Instances

X•IPC permits multiple instances to be started and to exist concurrently. In this manner it is possible to have multiple applications running, each involved with its own *X•IPC* instance. Multiple active *X•IPC* instances are completely segregated from one another.

The ability to define and start multiple *X•IPC* instances provides significant software management, development and maintenance benefits:

☐ It is easy to segregate projects and applications running on a single processor or over a network. Using *X•IPC* instances, the IPC resource requirements of each application are drawn from the application's own local private pool of IPC resources, instead of from some operating system's global pool of IPC resources. This ensures IPC resource availability for each application, without the need to constantly monitor the system-wide IPC pool for usage and contention.

☐ It is possible to run development and production versions of a system concurrently on a *single* processor or network. Development and support can occur side-by-side on one machine or network without any compromises or special adjustments.

☐ *X•IPC* instances are independent of one another. Distinct instances for each application lighten the management task of allocating IPC resource identifiers. As an example, each application can create and use its own `Test_Queue` without ever colliding with some other application's identically named queue.

☐ Finally, the segregation of instances guarantees that the activities within one instance have no impact on another. For example, debugging of problems in one instance (perhaps due to abusive use of *X•IPC* resources) has no effect on other active *X•IPC* instances.

4.10 Stand-Alone Instances

An *X•IPC* instance can be confined to a single processor in one of two ways. One approach is to start such an instance as a stand-alone instance. (The second approach, using a local instance, is described in the next section.

A stand-alone instance defines an *X•IPC* environment that is accessible by local processes only. Processes on other machines have no access to such an instance.

Because an *X•IPC*/stand-alone instance is *not* named or registered in any manner within any *X•IPC* naming catalog, it is ideal for establishing an *X•IPC* instance that is:

• inaccessible from any remote node

• invisible (except to programs that use it) within the node on which it is running

• used by intra-nodal *X•IPC* applications where no networking is involved.

4.10.1 INSTANCE NAMING

As was shown above, an instance that is local to one machine is identified by its instance configuration file name.

Example:

```
xipcstart  /home/sys/email
```

The above command starts the instance described by the "/home/sys/email.cfg" file. Had the "email.cfg" file been in the current directory, the following command would have had the same effect.

Example:

```
xipcstart  email
```

Each active instance is anchored to its host platform through its instance configuration file.

4.10.2 CONFIGURATION

The basic configuration sections `[XIPC]`, `[MOMSYS]`, `[SEMSYS]`, `[QUESYS]` and `[MEMSYS]`, as described in the respective subsystem Reference Manuals, are sufficient for starting a stand-alone instance. If a `[REMOTE_USER]` (formerly `[NETWORK]`) section appears in the instance configuration file, it is ignored.

4.10.3 ENVIRONMENT

The only environment variables used in conjunction with a stand-alone *X•IPC* instance are `XIPCROOT`, which is required at all times, and `XIPC`. When set, `XIPC` is assumed to contain the Instance File Name of the instance to be worked with. *X•IPC* commands requiring an Instance File Name as an argument refer to the `XIPC` environment variable when the Instance File Name argument is omitted from the command invocation.

4.10.4 STAND-ALONE COMMANDS

The following commands are used to start, stop and view an *X•IPC* stand-alone instance. .

4.10.4.1 xipcstart

`xipcstart` is used *without* the "-l" flag or "-n" flag which denote local or network instances. The instance started is based on the Instance File Name specified as an argument. If no Instance File Name argument is given, `xipcstart` uses the value of the `XIPC` environment variable.

4.10.4.2 xipcstop

The instance stopped is based on the Instance File Name specified as an argument. If no Instance File Name argument is given, `xipcstop` uses the value of the `XIPC` environment variable.

4.10.4.3 momview, queview, memview and semview

The instance monitored is based on the Instance File Name specified as an argument. If no Instance File Name argument is given, the monitor program uses the value of the `XIPC` environment variable.

4.10.5 PROGRAMMING TO ACCESS A STAND-ALONE INSTANCE

4.10.5.1 Environment

The `XIPCROOT` environment variable is required by programs that access a stand-alone *X•IPC* instance.

4.10.5.2 Logging into a Stand-Alone Instance

The XipcLogin() function identifies the target local instance by means of its Instance File Name.

Example:

```
XipcLogin ("/home/sys/email", "startup");
```

4.10.5.3 Program Linking

Programs that are to access a stand-alone *X•IPC* instance may be linked using either the *X•IPC* Stand-Alone library or the *X•IPC* Combined library. The topic of library selection is discussed in detail in the Advanced Topics section of this Guide.

4.11 Local Instances

An *X•IPC*/local instance is an *X•IPC* instance that is named, but whose name is only visible within the bounds of the node on which it is started.

An *X•IPC*/local instance is ideal for establishing an *X•IPC* instance that is:

- inaccessible from any remote node

4.11.4.1 xipcstart

`xipcstart` is used with the "-l" flag which denotes a local instances. The instance started is based on the Instance Local Name specified as an argument. If no Instance Local Name argument is given, `xipcstart` uses the value of the `XIPC` environment variable. Some examples follow:

```
xipcstart  /home/sys/email  -l EMail

xipcstart  -lEMail  /home/sys/email

xipcstart  -l EMail  /home/sys/email
```

4.11.4.2 xipcstop

The instance stopped is based on the Instance Local Name specified as an argument. If no Instance Local Name argument is given, `xipcstop` uses the value of the `XIPC` environment variable. An examp le follows:

```
xipcstop  /home/sys/email
```

4.11.4.3 momview, queview, memview and semview

The instance monitored is based on the Instance Local Name specified as an argument. If no Instance Local Name argument is given, the monitor program uses the value of the `XIPC` environment variable.

4.11.5 PROGRAMMING TO ACCESS A LOCAL INSTANCE

4.11.5.1 Environment

The `XIPCROOT` environment variable is required by programs that access a local *X•IPC* instance.

4.11.5.2 Logging Into a Local Instance

An XipcLogin() function call that is targeting a local instance specifies the instance by its local name with an "@" prefixed to it.

Example:

```
XipcLogin ("@EMail", "InitPgm");
```

In the above example, a program identifying itself as "InitPgm" logs into the local instance "EMail."

4.11.5.3 Program Linking

Programs that are to access a local *X•IPC* instance may be linked using the *X•IPC* Combined library. The topic of library selection is discussed in detail in the Advanced Topics section of this Guide.

4.12.3 NETWORK INSTANCE LOCATION

When working within an *X•IPC*/Network environment, it is possible to have multiple instances concurrently active on the network. Each active instance is physically located on the network node where it was started.

It is possible to have some nodes supporting more than one instance and others supporting no instances.

Of course, processes using a network instance have *no concern* for the instance's physical location since they refer to the instance by its network name.

4.12.4 NETWORK INSTANCE SEARCH RANGE

X•IPC commands and programs working within a network instance locate the physical position of the target instance as part of their instance login procedure. The range of machines searched is referred to as the *instance search range*.

The instance search range can be set in one of three ways:

☐ By explicitly specifying the instance's node name in the XipcLogin() call (demonstrated below).

☐ By specifying the name of one or more hosts (network nodes) where the instance should be searched for via the XIPCHOST and XIPCHOSTLIST environmental variables.

☐ By specifying the name of one or more *Catalog Nodes* where *X•IPC* maintains a catalog of network instances, via the XIPCCAT and XIPCCATLIST environmental variables.

Controlling the search range is accomplished using the following environment variables: XIPCHOST, XIPCHOSTLIST, XIPCCAT and XIPCCATLIST. Each program can set and control its own search range, using these variables. The order in which instance search range specification parameters are used follows:

1. The XIPCHOST environment variable.

2. The XIPCHOSTLIST environment variable.

3. The XIPCCAT environment variable.

4. The XIPCCATLIST environment variable.

5.   The environment variable XIPCCATLIST.

When more than one search specification is present, *X•IPC* uses the first one in the order listed above and ignores the rest.

4.12.11 NETWORK COMMANDS

The following commands are used to start, list, stop and view instances in a networked environment.

4.12.11.1 xipcstart

Starting an instance that is to be used over a network requires that an Instance Network Name be specified as part of the xipcstart command.

Example:

```
xipcstart   /home/sys/email   -n EMail
```

The above command starts an instance defined by the "/home/sys/email.cfg" instance configuration file and attaches "EMail" to it as its Instance Network Name.

Other possible forms of the same command include:

Example:

```
xipcstart   /home/sys/email   -nEMail

xipcstart   -nEMail   /home/sys/email

xipcstart   -n EMail   /home/sys/email
```

Were an XIPC environment variable set to "/home/sys/email," then the command could have been reduced to:

```
xipcstart   -nEMail
```

The specified Instance Network Name must be unique within the prescribed search range.

When xipcstart is invoked with the "-n" flag for starting a network instance, it searches the network for the existence of an active instance having the specified network name. If such an instance is found, the xipcstart command fails.

It is also possible to start an instance from under program control, using the XipcStart() function call. This function is described in the Advanced Topics chapter of this Guide and in the *X•IPC* Reference Manual.

4.12.11.2 xipclist - Listing Active Network Instances

It is occasionally important to know which network instances are active and where they are physically located. xipclist serves that purpose. It lists all active instances in the defined search range.Example:

```
xipclist
```

If a machine name is specified as an argument, then xipclist reporting is limited to that machine. In such a case, the search range is ignored.

In the above example, a program identifying itself as "InitPgm" logs into the network instance "EMail." The search for the "EMail" instance is conducted based on the settings of the `XIPCHOST,` `XIPCHOSTLIST,` `XIPCCAT` and `XIPCCATLIST` environment variables.

4.12.12.3 Program Linking

Programs that are to access a network *X•IPC* instance may be linked using either the *X•IPC* Network library or the *X•IPC* Combined library. The topic of library selection is discussed in detail in the Advanced Topics section of this Guide.

4.13 Multi-Instance Applications

In many cases, a one-to-one mapping scheme between an application and an *X•IPC* instance provides a sufficient level of abstraction for configuring and supporting the application's IPC requirements.

There are, however, situations––particularly when building larger applications––where it makes sense to split the application's IPC resources along certain physical or logical seams and to employ more than one instance for supporting the application's IPC activity. Such an application is a *multi-instance application*.

Issues related to the development of multi-instance applicationsare discussed in the Advanced Topics chapter of this Guide, in the section entitled "Working With *X•IPC* Instances."

5. *X⊥IPC* PROGRAMMING

5.1 Accessing An *X⊥IPC* Instance

5.1.1 XipcLogin() - LOGGING INTO AN INSTANCE

A user program must log into an instance before it can use its *X•IPC* environment. This is accomplished using the XipcLogin() function.

XipcLogin() takes the following arguments:

☐ The identity of the target instance.

☐ A login name by which the user will be known within the instance.

The target instance is specified in one of the following forms:

☐ Stand-alone instances are identified by their instance configuration file name.

Example:

```
RetCode = XipcLogin("/usr/demo", "myprog");
```

☐ Local and Network instances are identified by their Instance Local Name or Instance Network Name. An '@' character must be prefixed to the name to distinguish it from a stand-alone name.

Example:

```
RetCode = XipcLogin("@NetDemo", "myprog");
```

In the above examples, the calling program attempts to log into the instance named "NetDemo," using the login name "myprog."

Duplicate login names are permitted within an instance. It would thus be possible to have more than one user log in as "myprog" within the same instance.

XipcLogin() returns a non-negative instance "User Id" as its value when successful. This value is of minor significance and is generally not needed subsequently.

5.1.2 XipcLogout() - LOGGING OUT OF AN INSTANCE

A user logs out of an instance using XipcLogout(). XipcLogout() severs any connection between the calling user and the instance it is logged into.

XipcLogout() takes no arguments.

Example:

```
RetCode = XipcLogout();
```

XipcLogout() releases all held resources of the instance before it logs the user out. It is a good programming practice to have application program termination functions (such as trap handlers) call XipcLogout() before terminating.

The TIMEOUT Option

The `TIMEOUT` option is identical to the `WAIT` option except that, when blocking occurs, it is limited to the specified number of seconds of real time.

Should the blocked operation complete within the timeout period, the user is awakened and allowed to proceed. If, however, the stipulated time period expires, then the user's blockage is cancelled and an appropriate error code is returned by the function.

5.2.2 ASYNCHRONOUS OPTIONS

X•IPC additionally provides three asynchronous options for situations where it is desired that an operation complete in the "background." As such, it is possible–and often desirable–for a program to initiate multiple *X•IPC* operations that remain pending in the background until conditions permit them to complete.
The common denominator of the three asynchronous options is that the *X•IPC* operation does *not* cause the calling program to block. It continues unimpeded. The options differ in their method of completion notification. A key component of these approaches is the usage of a user-declared Asynchronous Result Control Block (ACB) variable. Each *X•IPC* operation that is directed to complete asynchronously has a user-specified ACB associated with it. The ACB allows the tracking (and possible aborting) of the operation if and when it blocks asynchronously. The ACB structure is additionally used by *X•IPC* for returning data from the asynchronous operation, when the operation completes.

Note that an *X•IPC* operation that is coded with an asynchronous option completes asynchronously whether or not it is forced to block before completing. This is the default behavior of the asynchronous options.

It is, however, sometimes required that an asynchronous option be applied *only* if the subject operation is forced to block, and to otherwise return synchronously if it can complete without blocking. This behavior can be accomplished by specifying the `RETURN` option flag together with the asynchronous option. Examples of using this option are given in the Advanced Topics section of this Guide.

The CALLBACK Option

The `CALLBACK` option directs *X•IPC* to notify of an asynchronous operation's completion by means of a user-defined callback function. The specified callback function is invoked when the blocked operation completes. The function's single argument is a pointer to the ACB associated with the completing operation. In this way, one function can be used to serve multiple asynchronous *X•IPC* operations.

The POST Option

The `POST` option directs *X•IPC* to mark the completion of the operation by setting a user-specified *X•IPC* event semaphore. The semaphore is set when the operation completes. A typical scenario would have another program or thread waiting for that semaphore to be set, and then to react appropriately. Alternatively, the semaphore can be examined at some later point in time by the original calling program or by another program in the instance.

The IGNORE Option

The `IGNORE` option instructs *X•IPC* to allow the operation to complete "silently." No explicit notification is given upon its completion. The original calling program may periodically poll the ACB associated with the pending operation, until the operation completes. Or it can ignore it entirely.

Refer to the specific function descriptions below and to the *X•IPC* Reference Manual for additional related descriptions. Further discussion of working with *X•IPC* 's asynchronous blocking options is presented in the Advanced Topics section of this Guide.

5.2.3 BLOCKING OPTIONS SUMMARY

The table below summarizes the uses of *X•IPC* 's synchronous and asynchronous blocking options.

structures. There are, however, rules that must be followed and understood in order to program multithreaded applications successfully.

This section describes general rules regarding the use of *X•IPC* within threaded programs. Operating system specific rules are delineated within the respective *X•IPC* Platform Notes. The reader is encouraged to refer to those notes after reading this general section.

5.3.1 *X•IPC* LOGIN PER THREAD

The first rule is that <u>each thread</u> must explicitly manage its own *X•IPC* logins, as it needs them. *X•IPC* logins <u>*cannot be shared*</u> across multiple threads. So, for example, an *X•IPC* program having five threads, each performing *X•IPC* operations, must be written so that each of the five threads performs its own XipcLogin() and XipcLogout() operations as necessary. In our example, that would translate into five separate logins. It is <u>not</u> <u>possible</u> for one thread to perform one login and have the context of that login shared by the five threads. (Of course, threads not doing *X•IPC* work need not log in to an instance.)

The "login per thread" rule has ramifications regarding *X•IPC* asynchronous programming. This is discussed below.

5.3.2 PROGRAMMING RESTRICTIONS

Following is a list of additional restrictions that are imposed on *X•IPC*–based threaded applications:

❑ The following three *X•IPC* list-building utility functions are *not* thread-safe: QueList(), SemList(), MemList(). These functions use internal static data areas which cannot be relied upon in a multithreaded environment. The other list building functions (e.g., QueListBuild(), QueListAdd(), etc.) are thread safe.

❑ Threaded programs, written for the supported **UNIX Operating System** platforms, cannot perform *X•IPC* operations that specify the *XXX*_TIMEOUT() blocking option.

❑ The MemSys function MemSection() is not thread-safe. One should use the MemSectionBuild() function instead.

❑ *X•IPC*, by default, cannot support more than 64 threads per process. In order to override this limit, one must set the external *X•IPC* variable XipcMaxThreads with the override value <u>before</u> the process performs its first XipcLogin() call.

5.3.3 ASYNCHRONOUS OPERATIONS

The XipcAsyncEventHandler() function that is called by an application upon completion of an asynchronous operation must be called by the <u>same</u> <u>thread</u> that issued the original *X•IPC* operation. It is possible to have another thread wait for the associated system event to occur (e.g., in UNIX to wait for the I/O descriptor to become set; in Windows NT to wait for the Event object to become set), but the final processing step of the operation – the calling of XipcAsyncEventHandler() – must be performed by the thread that initiated the original *X•IPC* operation.

UNIX-based multithreaded programs that issue *X•IPC* asynchronous operations are not able to receive notification of operation completion via system signals. Rather, they must use the I/O descriptor method of asynchronous notification. See the Technical Note "Using I/O Descriptors for Asynchronous Notification" for details on how this is programmed.

Each thread within a UNIX-based multithreaded program **must** set the XIPC_SETOPT_PRIVATEQUEUE option in order for its asynchronous operations to complete successfully. This is accomplished by having <u>each</u> thread call the XipcSetOpt() function, as follows, before it logs into *X•IPC*.

The above command starts the `semview` monitor for the SemSys of the "/usr/demo" instance. The initial update frequency is set to 100 milliseconds.

As was the case with `xipcstart` and `xipcstop`, the Instance File Name can be omitted from the command line. In such a case, `semview` also uses the value of the `XIPC` environment variable for the Instance File Name of the instance to start monitoring.

5.4.2 MONITOR FUNCTIONS AND LAYOUT

The *X•IPC* monitors are very similar in layout and function, sharing the same general "look and feel." Information is presented in a matrix-like display, where the users and the IPC entities make up the axes of the matrix. Interaction between users and IPC entities is displayed within the body of the matrix.

Asynchronously blocked *X•IPC* operations are treated as pseudo-users and receive an Asynchronous Uid (AUid) while they are pending. Information regarding AUids is displayed on the *X•IPC* monitors in the same manner as ordinary users. This provides a consistent means of monitoring pending asynchronous operations.

```
┌─────────────┬─────────────────────────────────┐
│  Monitor    │                                 │
│  Status     │       IPC Resources...          │
├─────────────┼─────────────────────────────────┤
│  Users      │                                 │
│   ...       │          Interaction            │
│   ...       │            Matrix               │
│   ...       │                                 │
│             │                                 │
├─────────────┼─────────────────────────────────┤
│  Command    │          Capacity               │
├─────────────┤                                 │
│Trace Operation│                               │
└─────────────┴─────────────────────────────────┘
```

Important subsystem capacity data is displayed at the lower right portion of the screen. Monitor status data is shown at the top left of the screen. The command entry window is at the lower left of the screen. The same format is used for all four subsystem monitors.

The trace window (located at the bottom of the command window) is active when the monitor is in one of the trace update modes (Flow or Step). It reports the next *X•IPC* operation to be executed and the Uid of the user performing the operation.

5.4.3 MONITOR MODES

X•IPC monitors operate in one of two modes:

☐ Update Mode

☐ Command Mode

5.4.3.1 Update Mode

When in Update Mode, the monitor updates the display of *X•IPC* activity in one of the following ways:

☐ "Interval Snapshot Mode" causes the monitor display to be refreshed at a user-specified interval rate (specified in milliseconds). Activity occurring between snapshots is not shown. This mode is useful for observing the general ebb and flow of activity occurring within an *X•IPC* instance.

```
Command> i2000
Command> i50
```

The first example sets interval to 2000 milliseconds, or two seconds. The second example sets "interval" to 50 milliseconds.

Very low interval settings will often cause performance degradation in the monitored instance subsystem.

5.4.4.2 Entering Trace Flow Mode

Entering Trace Flow Mode is accomplished by entering:

```
Command> tf
```

5.4.4.3 Entering Trace Step Mode

Entering Trace Step Mode is accomplished by entering:

```
Command> ts
```

5.4.4.4 Exiting Trace Mode

Leaving either Trace Mode returns the monitor to Interval Snapshot Mode. This is achieved by entering the Trace Off command:

```
Command> to
```

5.4.5 ZOOMING

X•IPC monitors provide a set of facilities for examining aspects of an instance with additional scrutiny. One of these tools is the monitor "zoom window." The other facility, the "browse screen," will be described shortly.

The monitor Zoom Window allows the developer to watch general *X•IPC* instance activity and at the same time focus on the activity of a specific aspect of the instance subsystem being viewed.

```
┌──────────┬──────────────────────────────────────┐
│ Monitor  │                                      │
│ Status   │         IPC Resources...             │
├──────────┼──────────────────────────────────────┤
│ Users    │                                      │
│   ...    │            Interaction               │
│   ...    │             Matrix                   │
│   ...    │                                      │
├──────────┴──────────────────────────────────────┤
│                    Zoom                          │
│                   Window                         │
├───────────────────┬──────────────────────────────┤
│    Command        │         Capacity             │
├───────────────────┤                              │
│ Trace Command     │                              │
└───────────────────┴──────────────────────────────┘
```

Each of the subsystem monitors provides a wide array of Zoom Window options from which to choose. The specific option codes and their applications are outlined in the subsystem specific sections below and in the *X•IPC* Reference Manual.

5.4.9 PANNING

As we have seen, *X•IPC* monitors are matrix-like in their layout. As such, they can be manipulated as on a spreadsheet when certain "off the screen" portions of the matrix are required for viewing. This is accomplished by Panning.

Panning can be performed horizontally or vertically. The exact format of the panning commands are subsystem specific and are provided in the susbsystem volumes.

5.4.10 EXITING THE MONITOR

Monitoringof an *X•IPC* instance is terminated using the q command. This is true for all subsystem monitors: momview, queview, semview and memview.

Example:

```
Command> q
```

Of course, bringing down an *X•IPC* monitor has no effect on the underlying instance, its ongoing activities or its client programs.

When monitoring is off, there is virtually no overhead to the performance of the instance (one additional machine instruction per *X•IPC* operation). This removes any need for building separate "debug" and "production" versions of a system. *X•IPC* production systems are automatically subject to *X•IPC* monitoring, even out in the field. It is thus possible for technical support personnel to remotely log into installed systems for analysis purposes using *X•IPC* monitors, if and when there are problems.

5.5 *X⊥IPC* Function Return Codes - Using XipcError()

X•IPC functions return negative integer codes whenever they do not complete successfully. These codes and their interpretations are described in the *X•IPC* Reference Manual and in appropriate sections of the subsystem-specific documentation.

By testing for a negative return value, it is immediately possible to check on a function's success or failure.

The XipcError() function is used for translating an error code returned by a failed *X•IPC* function call.

XipcError() takes one argument:

☐ The *X•IPC* error code whose translation is desired.

XipcError() returns a pointer to a static string containing a brief translation of the error code it is passed. It returns a pointer to an appropriate message for undefined error codes.

Example:

```
RetCode = QueCreate(...);

if (RetCode < 0)
{
    /* Error handling code */
    printf("QueCreate Error: %s\n", XipcError(RetCode));
}
```

6. ADVANCED TOPICS

This section of the *X•IPC* User Manual presents in-depth discussions of several advanced topics that can be central to optimizing your use of *X•IPC*. Most of the topics are presented from the perspective of the QueSys, SemSys and MemSys subsystems. Advanced Topics that are especially pertinent to MomSys are presented in the MomSys User Guide and MomSys Reference Manual.

6.1 Advanced Instance Configuration

This section describes instance configuration parameters that can be employed for making an *X•IPC* application take advantage of the specific hardware and operating system environment that it is running on.

The behavior of an *X•IPC* instance – and consequently of applications using the instance – can be influenced by a number of platform resources. The most critical platform resources involved are:

❑   The number of CPUs (processors) running on the platform

❑   The manner in which the instance's underlying memory sharing is implemented

X•IPC provides instance configuration parameters that can be used for influencing how an instance uses these resources.

6.1.1 CONFIGURING *X•IPC* FOR MULTIPLE-CPU (SMP) SYSTEMS

An *X•IPC* instance that will run on a multi-CPU platform such as a Symmetric Multiprocessor (SMP) computer should be configured differently than a single-CPU platform. The parameter that is involved is the CSEC_ALGORITHM instance configuration parameter. This parameter, found within the [XIPC] section, has two alternative values: Gate and Semaphore. (Some platforms support only one of these values, however; see the Platform Notes regarding your particular platform.)

The *usual* default value for CSEC_ALGORITHM is Gate . This setting is optimal for single-CPU systems. Multi-CPU systems should have their instances configured with CSEC_ALGORITHM set to Semaphore to override the default value.

Example :

```
[XIPC]
CSEC_ALGORITHM   Semaphore
```

The alternative value, Semaphore, can in certain circumstances be the default value. On certain UNIX platforms (e.g., HP-UX, Solaris and AIX 4.1 and higher), *X•IPC* is able to detect whether the underlying hardware is an SMP, or not. If it detects more than one processor active, then the CSEC_ALGORITHM parameter is set to a default value of Semaphore, which can be overridden to Gate. See the Platform Notes for details.

6.1.2 CONFIGURING AN INSTANCE'S MEMORY UTILIZATION

An *X•IPC* instance uses the operating system's underlying memory resources for supporting the activities of the instance. Exactly how this is accomplished depends on the operating system involved. The following table summarizes the *default* mechanisms used for implementing an instance's memory sharing requirements.

6.1.2.2 Configuring to Use Memory Mapped Files on UNIX

X•IPC instances that run on UNIX have the added flexibility that they may be configured to employ memory-mapped files instead of shared memory as the mechanism for supporting the instance's memory sharing elements. This is useful in situations where the UNIX system places size limitations on the size of shared-memory segments that can be created, thus inhibiting the size of the instances that can be run.

The `MAPFILE_CTL` and `MAPFILE_POOL` configuration parameters are used for specifying that the instance should use memory-mapped files instead of shared memory. The parameter is configured by specifying a file-system path name for the memory-mapped file that is to be created. The `MAPFILE_CTL` parameter defines where the shared control data should be created. The `MAPFILE_POOL` is only relevant for subsystems that have a text pool.

Example :

```
[QUESYS]
MAPFILE_CTL     /usr/harvey/quesys.ctl
MAPFILE_POOL /usr/harvey/quesys.pool
```

The following table lists the map-file configuration parameters that are valid in the respective instance configuration sections.

Section	Memory-Map Parameters (*UNIX only*)
`[XIPC]`	`MAPFILE`
`[MOMSYS]`	*(Not Supported)*
`[QUESYS]`	`MAPFILE_CTL, MAPFILE_POOL`
`[SEMSYS]`	`MAPFILE_CTL`
`[MEMSYS]`	`MAPFILE_CTL, MAPFILE_POOL`

Note that the configuring `MAPFILE` within the `[XIPC]` section is only valid when the `SHARED_MEM` parameter is also specified in that section. Such a configuration directs *X•IPC* to configure the entire instance as a single memory element, where the memory element is to be implemented as a memory mapped file.

Example :

```
# In the following example all subsystems within the instance
# consolidate into a single element that is implemented as a
# memory mapped file.

[XIPC]
SHARED_MEM      SINGLE
MAPFILE         /usr/projects/foo/xipc.ctl

[QUESYS]

[MEMSYS]
```

6.2 Asynchronous Operations

6.2.1 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, SEM_ER_ASYNC or MEM_ER_ASYNC return code, as appropriate. It is important to note that flags must always be ORed *to the left* of (before) the blocking option.

6.2.2 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.

```
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;
        }
```

```
            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;

    }
    Api;

};
```

6.2.3 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(), QueRead(), 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.

6.2.4 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.

The preceding 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. In that case, the application could utilize the various ACB User Data fields to distinguish between the pending operations as they complete. One possibility would be to assign an identifying code to each ACB, using one of the User Data fields.

```
/*
 * 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.

6.2.6 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:

```
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);
}
```

6.2.8 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.

6.2.9 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.

6.4 Working With *X•IPC* Instances

Central to any application built using *X•IPC* is the role of *X•IPC* instances. This section describes the issues related to working with *X•IPC* instances, as they relate primarily to the QueSys, SemSys and MemSys subsystems. Refer to the <u>MomSys</u> <u>User</u> <u>Guide</u> for detailed discussion of MomSys' utilization of *X•IPC* instances.

6.4.1 *X•IPC* INSTANCES: THE APPLICATION PERSPECTIVE

The role of instances in an *X•IPC*-based application can be understood from an *application* perspective and from a *process* perspective, as described in the following sections. At a high level, *X•IPC* instances are building blocks for defining an application's IPC environment. This environment can be monolithic or it may be divided into a number of parts. Strictly speaking, an application built using *X•IPC* is comprised of one or more *X•IPC* instances.

6.4.1.1 Single-Instance Applications

In many situations the level of abstraction provided by one *X•IPC* instance is sufficient for an application's IPC requirements. In such situations there is a one-to-one relationship between application and instance. Such an application is a *single-instance application*. The following diagram schematically presents an example of a single-instance application.

Of course, there is no restriction on the number of single-instance applications that can coexist on a single node or on a network, nor is there any restriction as to the type of instance (stand-alone or network) used in such applications.

Single-instance applications are the most basic way of using *X•IPC* to provide segregated IPC environments. Multiple applications sharing a machine or a network are automatically insulated from one another at the IPC level.

The multi-instance application model broadens the class of applications requirements that can be met using *X•IPC*. A large application having a hierarchical structure of processes can have its IPC environment constructed along similar hierarchical lines.

The actual physical positioning of the application's IPC components can then be determined in a manner that addresses the application's topological characteristics The following diagram offers two alternatives for positioning the previous application's processes and *X•IPC* instances. Either selection would have no effect on the *X•IPC* portion of the application.

Application architects may further employ the multi-instance application model to incorporate varying degrees of redundancy within their application's IPC environment. Critical elements of the IPC environment may be duplicated by using multiple instances in a primary (I) and backup (I') role. By duplicating its IPC activity, the application can be designed to recover if an outage occurs on the primary instance platform.

In fact, a process may log into an instance more than once, with each XipcLogin() operation establishing an independent logical session having a unique Uid within the instance. Examples of this appear later in this section.

A login session between a process and an instance may be expressed schematically as:

or algebraically as:

$$login = (I, \; Uid)$$

Thus, for example, when a process successfully logs into instance "xyz" and is assigned a Uid of 5 within the instance, this can be expressed as:
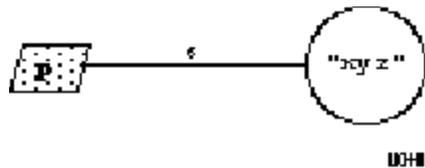
$$login = ("xyz", \; 5)$$

Processes that are part of a single-instance application will generally establish one login session with that application's one instance.

6.4.2.2 A Process's Working Set of Logins

Basic utilization of the *X•IPC* toolset typically involves processes that log in to a single *X•IPC* instance. This, in fact, is a limited usage of a much broader capability.

As indicated above, a process may establish multiple login sessions with a single instance or, for that matter, with multiple instances. Such is usually the situation regarding processes in a multi-instance application. Processes there will typically log into more than one of the application's instances.

The above notation is extremely useful for describing relationships between user processes and *X•IPC* instances.

6.4.2.3 Some Examples

The following examples essentially cover the gamut of possible cases:

☐ *Not Logged In Anywhere*

A process that is not logged into any instance, such as when it first starts executing, is defined as having a working set of logins that is empty:

$$L = \{\ \}$$

☐ *Multiple Logins/Multiple Instances*

An example of a process that has logged into multiple instances one or more times is:

$$L = \{\ (\text{"xyz"}, 2), (\text{"xyz"}, 3), (\text{"xyz"}, 10), (\text{"abc"}, 20), (\text{"qrs"}, 30)\ \}$$

☐ *Single Login/Instance*

A process that logs into mu ltiple instances, one login per instance:

6.4.2.4 A Process's Current Login

A process that has established multiple login sessions is actually connected to at most one of the logins at any point in time. That login session is referred to as the process's *Current Login*. *X•IPC* function calls that access and/or manipulate *X•IPC* objects do so using the context of the calling process's current login (i.e., its instance and Uid). As such, a process's current login defines, for that process, the instance being dealt with and the Uid being used within that instance, when *X•IPC* function calls are issued. It is thus generally an error for a process to execute an *X•IPC* function call while its current login is not defined. This is elaborated on below.

Consider the following example. Process P has established login sessions with two *X•IPC* instances. Two login sessions (Uids 3 and 14) are with instance "xyz." A third is with instance "abc." The process's current login is highlighted.

$$L = \{ \ (\text{"xyz"}, \ 3), \ (\textbf{"xyz"}, \ \textbf{14}), \ (\text{"abc"}, \ 5) \ \}$$

$$current\_login = (\text{"xyz"}, \ 14)$$

A process can control which login (from its working set of logins) is its current login by means of calls to the XipcLogin(), XipcLogout(), XipcConnect() and XipcDisconnect() functions. This is now demonstrated.

6.4.2.5 Modifying a Process's Working Set of Logins, Current Login

As was mentioned earlier, a process's working set of logins is initially empty:

$$L = \{ \ \}$$

In addition, the process's current login is initially undefined:

$$current\_login = ?$$

The means for adding and deleting logins from the working set and for setting the current login is by calls to XipcLogin(), XipcLogout(), XipcConnect () and XipcDisconnect(). The best vehicle for describing how these functions are employed for such activity is to present an example.

Functional descriptions follow for the XipcConnect() and XipcDisconnect() function calls as well as supplementary descriptions of XipcLogin() and XipcLogout(), describing how they affect a process's working set of logins and current login.

6.4.2.5.1 XIPCCONNECT() - Connect to a Login Session

The XipcConnect() function call sets the calling process's current login to the login session specified by the function's arguments. It is an error to call XipcConnect() when the process's current login is defined. In such a situation the process's current login must first be reset either by a call to XipcDisconnect() or by a call to XipcLogout(). These functions are described below.

XipcConnect() takes the following arguments:

☐ The instance name of the targeted login.

☐ The Uid of the targeted login.

The two arguments passed to XipcConnect() uniquely identify the login to connect to. Recall that an *X•IPC* login session is connoted as the pair (*Instance*, *Uid*). The specified login must be a member of the calling process's working set of logins.

Example:

```
Uid = XipcLogin ("@xyz", "AnyUser");
XipcDisconnect();
     ...
     ...
XipcConnect ("@xyz", Uid);
```

6.4.2.5.2 XIPCDISCONNECT() - Disconnect from the Current Login Session

The XipcDisconnect() function call resets the calling process's current login, leaving it undefined. It is an error to call XipcDisconnect() when the process's current login is not defined.

It is generally an error to issue an *X•IPC* function call when a process's current login is undefined. The exceptions to this rule are:

☐ XipcConnect() - to set the current login to an existing login session

☐ XipcLogin() - to establish a new login session, setting the current login to the new login

☐ XipcStart() - to start an *X•IPC* instance

☐ XipcStop() - to stop an *X•IPC* instance

☐ XipcInfoLogin() - to query information about the process's working set of logins. This function is described below.

XipcDisconnect() takes no arguments.

Example:

```
Uid = XipcLogin ("@abc", "AnyUser");
XipcDisconnect();
```

The basic interface to the XipcInfoLogin() function is presented below, followed by the more important interface which uses predefined macros as arguments.

XipcInfoLogin(), when used in its basic form, takes the following arguments:

☐ A pointer to a user-declared structure (or perhaps to the first element of an array of structures) of type XIPCINFOLOGIN, in which the requested login information is returned. This data type is described below. If this argument is NULL, then the function returns the number of elements currently in the working set of logins.

☐ The number of entries in the specified array.

☐ A pointer to a cursor variable used by *X•IPC* when using XipcInfoLogin() to enumerate the elements within the working set of logins. If the cursor pointer variable is NULL, information about the calling process's current login is returned.

XipcInfoLogin() generally returns as its value the number of login information structures filled and returned by the function call. XipcInfoLogin() returns a zero when no login information is returned.

A high-level interface to the XipcInfoLogin() function is provided by means of predefined macros that may be used as arguments to the function:

☐ The XIPC_LOGIN_CURRENT macro may be used to access information about the calling process's current login.

Example:

```
XIPCINFOLOGIN InfoLogin;

XipcInfoLogin (&InfoLogin, XIPC_LOGIN_CURRENT);
```

When called using the XIPC_LOGIN_CURRENT macro, the function returns information about the calling process's current login within the XIPCINFOLOGIN structure.

☐ The XIPC_LOGIN_COUNT macro may be used for accessing the size of the calling process's working set of logins.

Example:

```
XINT SetSize;

SetSize = XipcInfoLogin (XIPC_LOGIN_COUNT);
```

Example:

```
/*
 * The following sample program demonstrates a mechanism for performing
 * queue dispatch operations within a multi-instance XIPC environment.
 * The application's XIPC environment involves three instances: "abc",
 * "xyz" and "qrs". The application accesses two message queues named
 * "EMailQueue" and "DataBaseQueue". The application is not required to
 * know in which instance each queue is.
 */

VOID
main()

{

    CHAR    *I1 = "@abc";
    CHAR    *I2 = "@xyz";
    CHAR    *I3 = "@qrs";
    CHAR    *Name = "Example";
    XINT    Uid1, Uid2, Uid3;


    Uid1 = XipcLogin (I1, Name);
    XipcDisconnect ();

    Uid2 = XipcLogin (I2, Name);
    XipcDisconnect ();

    Uid3 = XipcLogin (I3, Name);
    XipcDisconnect ();

    SendMsg ("EMailQueue",   "This is an EMail message",   100);
    SendMsg ("DataBaseQueue", "This is a database message", 200);

    XipcConnect (I1, Uid1);
    XipcLogout ();

    XipcConnect (I2, Uid2);
    XipcLogout ();

    XipcConnect (I3, Uid3);
    XipcLogout ();
}
```

This approach can be employed to provide transparent multi-instance access to *X•IPC* semaphores and shared-memory segments as well. Enhancements can be added to optimize for situations where repeated accesses are to occur, such as building a table of accessed objects within the FindQueue() function. One approach is included as part of an example that is presented later in this section.

6.4.2.7 Asynchronous Operations in a Multi-Instance Environment

Ongoing *X•IPC* asynchronous activity related to a particular process is *not* affected by the current login setting of that process.

Specifically, a process working within a multi-instance environment may initiate numerous asynchronous operations within these instances and, when the operations complete, the process will be notified of each completion in the manner that was specified when the operation was started (i.e., callback function, post semaphore or ignore), regardless of the process's current login at the time the operation completes.

This can be described by means of the following diagram. Process P is currently logged into three instances "xyz", "abc" and "qrs" (twice to "xyz"). The process has initiated a number of asynchronous *X•IPC* operations in the course of its work within the three instances. Perhaps it is waiting asynchronously for certain events to occur, or for certain messages to arrive within those instances. It is currently connected to login ("qrs", 3). Otherwise stated, the login ("qrs", 3) is the process's current login.

The process will be notified of each asynchronous operation completion as it occurs within any of the three instances, regardless of the fact that the process is currently connected to a login session within instance "qrs." In fact, it would work as well if the process had not been connected to any of its login sessions at the time that the asynchronous operation(s) completed.

Essentially, notification of asynchronous *X•IPC* events is passed to a process regardless of the process's current login status.

As an example of this concept, consider a modified version of the previous example where the process now issues asynchronous QueReceive() operations using queues that are defined within the multi-instance environment.

Example:

```
/*
 * The following sample program demonstrates a mechanism for performing
```

```
    XipcConnect (I3, Uid3);
    XipcLogout ();

}


XINT
RecvMsgAsync (QueueName, MsgBuf, MsgBufSize, CallBack, Acb)
CHAR *QueueName;
CHAR *MsgBuf;
XINT MsgBufSize;
VOID (*CallBack)();
ASYNCRESULT *Acb;

{
    /*
     * This function calls FindQueue() to connect to the login session
     * whose instance contains the desired queue. It then issues the receive
     * operation. It then disconnects the process from the login.
     */

    XINT    Qid;
    XINT    RetQid;
    XINT    Priority;

    Qid = FindQueue (QueueName);

    QueReceive (QUE_EA,
                QueList(Qid, QUE_EOL),
                MsgBuf,
                MsgBufSize,
                &Priority,
                &RetQid,
                QUE_CALLBACK(CallBack, Acb));

    XipcDisconnect();

}
```

6.5 Starting and Stopping Instances Under Program Control

6.5.1 XipcStart() - STARTING AN INSTANCE

The XipcStart() function call is used for starting an *X•IPC* instance**Error! Reference source not found.** from within a program. This form of instance control is needed for situations in which using the xipcstart command is not appropriate.

The XipcStart() function call takes the following arguments:

☐ The Instance File Name of the instance to be started. Recall that the Instance File Name identifies the configuration file (excluding the ".cfg" extension) to be used when starting the instance.

☐ The Instance Name (Local or Network) to be assigned to the instance. In the case that the instance is being used in a stand-alone environment, this parameter must be set to NULL. If NULL, the name will be taken from the parameter specified in the [XIPC] section of the Instance Configuration File If no naming parameters are specified within the .cfg file either, then the instance is started as a Stand-Along instance having no registered name. Such an instance is only accessible via its Instance File Name. (See the *X•IPC* Reference Manual for further information.)

☐ An Options parameter to indicate reporting, testing, initializing and other options.

Example:

```
/*
 * Start an instance that is based on the
 * "/projects/tpsys.cfg" configuration file.
 * Assign it the network name: "TPSYS".
 * The startup report is generated on 'stdout'.
 */

 XipcStart ("/projects/tpsys", "TPSYS",
XIPC_START_REPORT|XIPC_START_NETWORK);
```

Example:

```
/*
 * Start the same instance as in the previous example,
 * but this time as a stand-alone instance. Also, suppress
 * the startup report.
 */

 XipcStart ("/projects/tpsys", NULL, 0);
```

XipcStart() creates the IPC environment as described within its Instance Configuration File. As such, it must be called before any program can log into the instance and use its IPC environment.

The XipcStart() function will only succeed when called as part of an *X•IPC*/Stand-Alone program to start a local instance. It will otherwise return XIPC_ER_NOTLOCAL.

6.6 Using *X•IPC* Libraries

6.6.1 INTRODUCTION

The *X•IPC* toolset includes a number of libraries for building applications using the *X•IPC* API. Technical instructions for using these libraries are generally platform-specific in nature and are therefore included as part of the <u>Platform Notes</u> for each of the platforms supported by *X•IPC.*

This section presents a high-level discussion of the issues relating to the usage of the different *X•IPC* API libraries, focusing on the advantages and disadvantages of using each of the libraries in various application settings. The term *library*, as used in this section, is applied in its generic sense. Some of the platforms that support the *X•IPC* toolset refer to collections of object modules using different terminology. For the sake of simplicity, this section will use the term *library*.

The *X•IPC* API library can be used in three modes, each of which addresses a specific application class. The three modes are generally referred to as:

☐ The *X•IPC* Stand-Alone Library

☐ The *X•IPC* Network Library

☐ The *X•IPC* Combined Library

The determining factor in deciding which library to use for linking a program is the program's intended proximity relative to the *X•IPC* instance(s) that it will be working with. Examples of different scenarios that cover the range of possible situations are presented below.

6.6.2 THE *X•IPC* STAND-ALONE LIBRARY

Consider the following situation:

The advantage of using the *X•IPC* Network Library is that:

- ☐ It produces a smaller executable than those produced by either the *X•IPC* Stand-Alone Library or the *X•IPC* Combined Library.

The disadvantage of using the *X•IPC* Network Library is that:

- ☐ The resulting executables can only interact with remote *X•IPC* instances.

6.6.4 THE *X•IPC* COMBINED LIBRARY

Maximum flexibility is achieved using the *X•IPC* Combined Library. Programs that are intended to interact with local and/or remote *X•IPC* instances should be linked using the *X•IPC* Combined Library. Such programs are said to belong to the class of *X•IPC* Combined programs.

Consider the following situation:

Program P1 is linked with the *X•IPC* Combined Library. It can therefore interact with either or both of instances I1 and I2, even though I1 is local and I2 is remote. In addition, the interaction between P1 and I1 is carried out in a manner as direct as if the program were linked using the *X•IPC* Stand-Alone Library (i.e., avoiding the network environment entirely).

The same P1 executable would work *without relinking* (assuming N1 and N2 are homogeneous platforms) in the following three situations:

6.6.5 CONCLUSION

The three types of *X•IPC* API Library provide a wide range of flexibility for using the *X•IPC* toolset under different application settings.

In most situations, linking with the *X•IPC* Combined Library is a safe choice–unless there is no network available whatsoever, in which case the *X•IPC* Stand-Alone Library is required. As an application evolves, its individual programs can subsequently be linked using either of the other *X•IPC* libraries to benefit from their advantages, where possible.

The `XIPC_TRAP_FUNCTION_TEST()` macro requires arguments that are operating system specific. Refer to the appropriate <u>Platform</u> <u>Notes</u> for details of its calling sequence.

As an example, consider the following body of a trap service function:

```
{

    /*
     * System call to reset the operating system 'signal'
     * or 'trap' flag should go here (if required).
     */

    XIPC_TRAP_FUNCTION_TEST( ... );

    /*
     * The remainder of the function can safely
     * service the asynchronous 'signal' or 'trap'.
     */

    ...
    ...
    ...

    return;
}
```

A note regarding trap service functions and the SemList(), QueList() and MemList() functions: Recall that these list functions build their lists in their own internal static memory. Calling these functions from within a trap service function is thus dangerous, since the interrupted process might have been in the middle of using this same static area.

It is therefore much safer to use the SemListBuild(), QueListBuild() and MemListBuild() functions instead, because they build their lists using user-specified list variables. These list variables (of type SIDLIST, QIDLIST or MIDLIST) should ideally be automatic (i.e., stack) variables; as such, they would avoid the above-stated problem.

Two important notes regarding XipcMaskTraps() are:

☐ It only prevents the complete execution of trap handling functions (assuming they are coded with the `XIPC_TRAP_FUNCTION_TEST()` macro at their start). It *does not* control whether signals arrive at the process. A process that can receive a signal while traps are masked should be coded to restart interrupted synchronous operations executed during that period.

☐ Asynchronous *X•IPC* operations that complete while traps are masked are prevented from having their completion processing performed (e.g., running a callback function), until the mask is lifted via a call to XipcUnmaskTraps().

It is therefore advisable to use XipcMaskTraps() and XipcUnmaskTraps() to mask traps for limited periods of time and only when necessary.

6.9 Extending *X⅃IPC* 's Functionality

X•IPC provides the developer with the means for extending *X•IPC*'s capabilities beyond its basic functionality. User-written functions, built upon the *X•IPC* API, can provide greatly expanded and specialized forms of IPC functionality.

Examples of extending *X•IPC*'s functionality could include user-written functions that:

☐ Increment a word of shared memory "atomically."

☐ Analyze the contents of all the messages on a message queue.

☐ Collect IPC statistics as part of a user-designed IPC monitoring system; collected data can be used for display purposes or for dynamic system intervention.

☐ Capture periodic images of message queue, shared memory contents or event semaphore settings.

6.9.1 INCREMENT A SHARED MEMORY WORD ATOMICALLY

Consider writing a user function that increments a four-byte "word" of *X•IPC* shared memory "atomically." The target memory word is to be identified by *Mid* and *Offset*. The function should return the value of the incremented word.

By masking MemSys traps and then freezing the subsystem, a series of MemSys operations can be issued that are guaranteed to be run as an "atomic" unit, without trap function interruption and without other MemSys user operations executing interwoven within.

This is a basic requirement for coding a user-defined atomic operation that issues multiple *X•IPC* function calls.

Example:

```
/*
 * MemIncr() --- Version 1.
 */

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

    /*
     * Stop everything.
     */

    XipcMaskTraps();
    MemFreeze();

    /*
     * Perform the necessary MemSys operations.
     */

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

```
     /*
      * Restart everything.
      */

     MemUnfreeze();
     XipcUnmaskTraps();
     return (Data);
}


/*.........................................................*/
/*
 * MemIncrLock() --- Performs increment operation using
 *                   MemLock and MemUnlock.
 */

XINT
MemIncrLock(Mid, Offset)
XINT Mid;
XINT Offset;
{
     SECTION TempSec, RetSec;
     MIDLIST MidList;
     XINT        Data;

     /*
      * Perform the MemIncr operation
      * using MemLock/MemUnlock to wait
      * for target to become accessible.
      */

     XipcMaskTraps();

     MemListBuild(MidList,
         *MemSection(&TempSec, Mid, Offset, 4L),
         MEM_EOL );

     MemLock (MEM_ALL, MidList, &RetSec, MEM_WAIT);
     MemRead (Mid, Offset, (CHAR *)&Data, 4L, MEM_NOWAIT);

     Data ++;

     MemWrite (Mid, Offset, (CHAR *)&Data, 4L, MEM_NOWAIT);
     MemUnlock (MidList, &RetSec);

     XipcUnmaskTraps();
     return (Data);
}
```

This version will perform like the first example, so long as the calling user has read/write access to the targeted memory area. If the area is found inaccessible by either MemRead() or MemWrite() then the operation is performed using a memory locking approach by a call to MemIncrLock().

To summarize, the ability to extend *X•IPC*'s functionality greatly broadens the range of IPC application requirements that can be addressed using the *X•IPC* product.

♦   `SemInfoUser()` - Provides user information;   also provides information about pending asynchronous operations and SemSys events.

Other secondary information verbs are provided as well for reporting less significant information occurring within the specified subsystem.

Using these verbs it is possible to build customized monitor processes within an application that oversee the internal operations of the application. It is additionally possible to build customized GUI-based application monitors that display data retrieved from these functions in a customized display format.

6.10.3 UNDERSTANDING X•IPC INFORMATION VERBS

Within the family of Information verbs, all of the verbs listed above can be employed to obtain information about a *series* of subsystem data items. The programming method for looping through the series of items in this group, using a MomSys example, is:

1.   Initially, call `MomInfoXxx(MOM_INFO_FIRST, &...)` .

2.   Subsequently, call `MomInfoXxx(MOM_INFO_NEXT(...), &...)` .

3.   Stop when the return code is `MOM_ER_NOMORE` .

Two other verbs in the MomSys subsystem -- `MomInfoAppQueueWList()` and `MomInfoUserAlist()` -- can be used to report more detailed MomSys information. The programming method looping through the series of items in this group is:

1.   Initially, call the corresponding `MomInfoXxx()` verb, and use its output parameter both to initialize a cursor variable (e.g., `MyCursor`) to the position of the first element of the *X*List, and also to obtain information about that element

2.   Then, call `MomInfoXxxXList(..., &MyCursor, &...)`. This advances `MyCursor` to the position of the next element of the *X*List and obtains information about that element.

3.   Stop when the return code is `MOM_ER_NOMORE`.

6.10.4 CODING EXAMPLES OF MOMSYS INFORMATION VERBS

The following two code templates illustrate the two styles of information-gathering loops (including error checking).

Example 1:

```
/*
 * Sample of MomInfoXxx()verb usage - e.g. for MomInfoAppQueue().
 * Loop through all the app-queues in the current instance,
 * retrieving and processing the status data of each app-queue.
 */

MOMINFOAPPQUEUE MyInfoAppQueue;
XINT            RC, MyAQid;

for (RC  = MomInfoAppQueue( MOM_INFO_FIRST, &MyInfoAppQueue );
     RC != MOM_ER_NOMORE;
     RC  = MomInfoAppQueue( MOM_INFO_NEXT( MyAQid ), &MyInfoAppQueue ))
{
    if (RC < 0)
    {
        /* Take appropriate error action for MyInfoAppQueue */
        . . .
        break;
```

If one wanted to loop through *all* the app-queues' wait-lists, then the second code segment above would be nested within the first segment, so that the processing of each app-queue would entail traversing its wait-list.

Refer to the respective Reference Manual pages for additional details on the usage of these verbs.

6.10.5 SAMPLE QUESYS FUNCTION

Consider the QUEINFOSYS data structure. A pointer to such a structure is passed as a parameter to the QueInfoSys() function for accessing status information about an instance's QueSys.

The QUEINFOSYS structure returns with assorted identification and statistical data that describe the status of the QueSys subsystem. QueInfoSys() additionally returns with *Wait List* data relating to the subsystem's Message Text Pool. The Wait List identifies the list of QueSys users currently blocked on QueWrite() operations to the Message Text Pool.

The QUEINFOSYS data type includes the following fields:

```
typedef struct
{
    /*
     * Identification and statistical data.
     */


    ...
    ...

    /*
     * List data.
     */

    XINT            WListTotalLength;
    XINT            WListOffset;
    XINT            WListLength;
    QUE_SYSWLISTITEM;   WList[QUE_LEN_INFOLIST];

} QUEINFOSYS;
```

where the `QUE_SYSWLISTITEM` data type is defined as:

```
typedef struct
{
    XINT            Uid;
    XINT            MsgSize;

} QUE_SYSWLISTITEM;
```

The *WListOffset* field of the QUEINFOSYS structure should be set before QueInfoSys() is called. Setting it to zero instructs the function to return with WList data, starting with the first element on the list.

Example:

```
/*
 * Report all blocked QueWrite operations
 * currently occurring in QueSys.
 */

QUEINFOSYS    InfoSys;
XINT          i;

/*
 * Set offset to zero so that WList data is returned
 * from the start of the list.
 */

InfoSys.WListOffset = 0;

do
{
    QueInfoSys ( &InfoSys );

    for (i=0; i < InfoSys.WListLength; i++)
        printf("Uid %d blocked on QueWrite of %ld bytes",
        InfoSys.WList[i].Uid, InfoSys.WList[i].MsgSize);

    InfoSys.WListOffset += InfoSys.WListLength;

} while (InfoSys.WListOffset < InfoSys.WListTotalLength );
```

The problem with issuing multiple calls to QueInfoSys() to report on Wait List status is one of data variability. Things can happen between the calls.

This can be prevented by freezing QueSys and masking traps for the duration of the reporting loop. This will guarantee that the collected Wait List data is complete, accurate and consistent.

The method outlined in the next example for acquiring Wait List data can be applied in a similar manner to all of *X•IPC*'s "Info" function list data.

In summary, by using the "Info" function calls in conjunction with the XxxFreeze() and XipcMaskTraps() function calls, it is possible to build customized IPC reporting and monitoring capabilities into your product, tailored to the specific real-time reporting needs of the application. An example of this approach follows:

Example:

```
/*
 * Report all blocked QueWrite operations
 * currently occurring in QueSys.
 */

QUEINFOSYS    InfoSys;
XINT          i;


/*
 * Stop everything.
 */
```

6.11 The *X•IPC* Command Interpreter

The *X•IPC* toolset provides the application developer with the ability to manipulate an *X•IPC* instance and its IPC objects interactively. This is accomplished using the `xipc` interactive command interpreter.

`xipc` provides an interactive interface to all the *X•IPC* API's. `xipc` is most useful in situations where access to an instance and its IPC objects are required on an ad-hoc basis. This need can arise throughout the application development process.

With `xipc` it is possible, for example, to:

- ☐ Create or delete *X•IPC* objects (queues, app-queues, semaphores, segments)

- ☐ Set or clear event semaphores

- ☐ Initialize the contents of a shared memory segment

- ☐ Extract arbitrary messages from a message queue

- ☐ Lock (or otherwise set the access privileges of) areas in shared memory

- ☐ Clean up after "untidy" user programs

- ☐ Insert messages onto one or more message queues

- ☐ Control a queue's overflow spooling activity

- ☐ Set or reset *X•IPC* triggers

- ☐ Initiate an asynchronous *X•IPC* operation that, when completed, executes another *X•IPC* command or native operating system command

- ☐ Log into a corrupted *X•IPC* instance to examine its status

- ☐ etc.

The `xipc` interactive interpreter is based on a language of commands. The exact syntax and details of using the `xipc` interactive command language are described in the chapter on *X•IPC* Commands in the <u>*X•IPC* Reference Manual.</u>

6.11.1 SAMPLE USAGE OF THE *X•IPC* INTERACTIVE COMMAND INTERPRETER

This section presents a selection of sample `xipc` sessions. The examples demonstrate the types of situations where using `xipc` can provide important time-saving development assistance.

Sample 4 : Log into two instances.

```
xipc> xipclogin /proj/xipc/demo Jack
      Uid = 11
xipc> xipcdisconnect
      RetCode = 0
xipc> xipclogin @DB Jack
      Uid = 15
xipc> xipcinfologin
      Uid   Instance
      ---   --------
       15   @DB
       11   /proj/xipc/demo
```

A brief note regarding this last example: The superuser capability, in providing a means for logging into an otherwise inaccessible instance, also returns the instance to a state where it may be accessed on a general basis (such as by the *X•IPC* monitor programs).

There is no guarantee, however, that the instance is in complete working order. The superuser login is most useful for ascertaining the identity of the program that last accessed the instance, i.e., the one that was the likely cause of the instance becoming corrupted.

7. INDEX