DATE: June 20, 1984

TO: LIST and R & D Personnel

FROM: Jerry Kazin, Joel Ball

SUBJECT: Inter-Process Communication Functional Specification

REFERENCE: PRIMOS Support For Premier Systems Proposal
PE-TI-1064
The New Print ServiceDesign
PE-TI-XXXX
Software Interrupt Control Module Functional Specification
PE-TI-1005
Specifications For The PRIMOS Condition Mechanism
PE-T-468, Rev. 2
Dynamic Storage Allocator Functional Specification
PE-TI-1062

KEYWORDS: IPC, Mailbox

Abstract

The need for an Inter-Process Communication (IPC) mechanism at the user level has long existed here at PRIME, but we have not yet provided such a facility. Most other commercially available operating systems provide their users with this functionality. This paper describes a prototype IPC which will be built into PRIMOS for internal use only.

This document has now been updated to reflect changes made at Primos release 20.0. Functionally, the IPC has not changed radically for 20.0. It has, however, changed drastically internally. Please refer to the IPC Design Specification for detailed information on the internal changes. The major addition at this release is the capability for remote IPC communications. There are some fine points about the use of IPC under remote conditions, but they will be indicated by a revision bar.

Table of Contents

1 FUNCTIONALITY

Inter-Process communication is a very important piece of functionality that any good operating system should provide its users. It covers a broad range of capabilities and functionality. Around PRIME its creation would permit certain operating system functions ease of implementation as well as allowing subsystems needed and desired flexibility. In addition, making IPC an available feature of our operating system would give many potential buyers of the system more reason to consider purchasing PRIMOS. Unfortunately, due to the lack of specifications relating to a distributed environment and a lack of time and manpower, the IPC mechanism described by this paper will not be released to our customers at this time.

This IPC will be usable across the network. This will be achieved through the use of NPX in conjunction with the use of remote pathnames. These pathnames will act as name space managers. Remote IPC is not available to pre-Rev 20.0 users, however, it will provide communications for processes local to each other on pre-Rev 20.0 systems.

The basic design criterion used in this proposal is simplicity: in use, in design, in interface, and in program operation.

It is important to remember that any IPC will not fill all the needs of all of its potential customers. In particular it should be again noted that this IPC is a prototype which will be replaced in the future by a fully specified distributed version, one that is capable of supporting the SROS environment. All users of this IPC will, at that time, have to change to the new ISC. We will try to implement the user interfaces as simply as possible to facilitate an easy conversion.

1.1 Terminology

The following defines all special terminology used in this specification. It is assumed that readers of this document are somewhat familiar with PRIMOS.

mailbox - The object through which the IPC passes data. It is a virtual object named by the user and dynamically built by the IPC mechanism.

software interrupt - Asynchronous interrupt generated by software which may be seen by a user's software. Terminal Quits are an example of a software interrupt.

condition - Name associated with a given software interrupt. The condition associated with terminal breaks is "QUIT$".

signal - Action performed when raising a condition. Signals are used as the agent to tell a user that a software interrupt has occurred.

on-unit - Software structure to which control is passed when signalling occurs.

1.2 Goals and Non-Goals

This IPC intends to provide certain internal PRIME users with enough of a cross process data passing and synchronization mechanism so as to enable limited communications. (Data passing and synchronization are the two most important features of any IPC.)

This project will make use of the prior work of the Network group, Lee Scheffler, and Jerry Kazin when designing both user and internal functions.

This IPC will not be directed towards solving the general system problem of IPC usage. We do not have enough knowledge nor time to provide that kind of solution.

This IPC will not solve the general problem of name space management. We will not build a name server.

This IPC will not provide any recoverability for messages lost when there is a system crash.

The design of this product, even with its limited capabilities, will be directed towards ease of use. The system should not require a thorough knowledge of IPCs before being available for programmer's use. It must be simple.

The IPC should make it easy for users to dynamically create the pathways through which the data will pass.

This IPC will provide the same security as the file system by utilizing existing PRIMOS functionality, (this is much the same concept as Named Semaphores).

The IPC should be well integrated into PRIMOS without interfering with any present resource. In other words, it should present itself to PRIMOS as a piece of functionality (black box) that can simply be added to the load. It will make use of known internal base level functions. It will cause minimal changes in the present product.

Finally, the IPC will be built and will accomplish its stated task in an acceptable and intelligent manner as regards both its user interface and its internal design.

1.3 The User's Model

The users of this IPC will be restricted to PRIME internal customers. These users must understand that this product may be replaced at any time by another version. They must be willing to take the responsibility to convert to the new ISC before choosing to use this IPC. We cannot guarantee that the program interfaces will stay the same.

The users of this IPC will initially be of the systems programmer category. Therefore, even though the product is being designed in as simple a manner to use as possible, some basic knowledge of the programmer's resourcefulness is assumed.

Three users of this IPC are presently known and are driving this project towards a timely completion. They are the New Print Service project, Premier Systems and HCR's Unix. All will be using the IPC to communicate between user and server or other user processes.

1.4 Product Functions

The following sections briefly discuss the functionality and limitations of this IPC.

1.4.1 Syntax/Form

The new IPC will be broken into two areas, a control mechanism and a data passing mechanism.

1.4.1.1 Control Mechanism

The control mechanism is the portion of the IPC used to allow access to and from the data passing mechanism and between processes. It will consist of four parts; a semaphore notification mechanism, an inter-process software interrupt mechanism, an access control method, and a programmable on/off switch for the interrupt mechanism.

1.4.1.1.1 Semaphore Notification Mechanism

Presently, PRIMOS contains a semaphore notification mechanism which allows the system to put users into a wait state. This is done for certain activites such as waiting for a character to be entered at a terminal or for waiting on a SLEEP$ timer. While in the wait state, a user process does not take any cpu time. Additionally, the semaphores used by this mechanism have been made software interruptable, i.e., the semaphore may be notified by an event other than the event for which the user is waiting. For example, in the case of waiting for the SLEEP$ timer to expire, the user may enter a break (terminal quit). This will cause the user to be awakened even though the timer did not expire.

This method of controlling when a user is waiting and when to awaken the user is said to be synchronous.

The IPC will use this semaphore notification mechanism to allow users to wait for messages to be sent to them.

1.4.1.1.2 Inter-Process Software Interrupt Mechanism

Software interrupts within PRIMOS are presented to a process via the CRAWLOUT mechanism. This mechanism follows the PL/1 Condition Mechanism standard. Conditions are generated by signals which may be intercepted by the programs with on-units. For the IPC, whenever a process is to be signalled that a message is waiting, the "IPC_MSG_WAITING$" condition will be generated for the

process. The process will then be able to intercept this condition with an appropriate on-unit. This on-unit then has the responsibility of retrieving the message from the IPC.

PRIMOS currently allows software interrupts to be generated in another processes' execution environment. This is done for forced logout, phantom logout notification, and other conditions. This mechanism will be used by the new IPC to tell the CRAWLOUT mechanism to notify the receiver of a message that a message is waiting. This means that a server will not need to poll to determine if data is waiting.

In addition, the server must always attempt to read all possible messages when a software interrupt (or semaphore notify for that matter) happens. It is possible for message notification to be lost or over notified. Therefore, it is the responsibility of the programmer to for see these events and program for them.

These interrupts may only be initiated from ring 0 for the obvious reason of security. Therefore, the new IPC must exist in the ring 0 environment.

This method of informing a user that a message is waiting is said to be asynchronous.

1.4.1.1.3 Access Control

Access control will be accomplished in a manner similar to Named Semaphores. An access category whose access rights have been set-up prior to the invocation of the servers will serve as the access control for the mailbox in question. That is, only users with correct rights as specified in the access category associated with the mailbox in question will be able to access the mailbox and/or the data in the mailbox.

Another way of stating that access catagories are used for access control is to state that the file system is being used for name space management. This means that the access category name will be used as the mailbox name. If the access category being used for name space management is moved, the program which uses this name must be updated accordingly, a drawback.

Additionally, the configuration of the network must be known to the programmers using the IPC as access catagories may be found on different nodes of the system and these nodes must be able to communicate. This means that the network disk configuration available to the local programmer is his local name space. He cannot communicate

with any user who cannot also see the same exact disk. This also implies that a mailbox is local to the system that owns the disk.

The only kinds of rights to place on an access category that make sense are read and write. The directory containing the access category may have add and delete rights. If these rights are present, a user will be allowed to dynamically create and/or delete an access category (mailbox) with its associated access rights. Without these rights, the mailbox may be thought of as being permanent.

Most often, systems using this IPC mechanism will set up the access catagories prior to the execution of the system. This means that the need to dynamically create and/or delete a mailbox as mentioned above will most often not be needed.

The rights assigned to a mailbox via its associated access category are only used during the mailbox open operation.

It will be possible for a user to have both read and write access to a mailbox. This means that mailboxes are two way objects. It also implies that a user may communicate with himself.

1.4.1.1.4 ON/OFF Switch

The on/off switch interface will make it possible for a server to turn on/off the asynchronous IPC message notification noted in the section on the Inter-Process Software Interrupt noted above. Any notification that might have happened while a process was in the off state will be remembered as pending. Please refer to section 7 for further details on the use of the on/off switch.

1.4.1.2 Message Passing

Data will be passed as message packets making this IPC a message based IPC. These packets will be automatically queued for the IPC users. Each packet will contain both IPC mechanism control information such as who is to receive this packet and the sender's data. Senders will not be told when a user receives a message. Each data queue defines a data pathway that only users with the proper access rights may see. Associated with the data pathway is a control block used to control quick access to the mailbox associated with the data pathway.

Mailboxes will have the capability to store the data sent to a specific user, many users, or all users. Data sent to any

user will be retrieved by the first user asking for data. Data sent to all users will be kept until all users have read the data. This may be referred to as a broadcast message.

No recoverability will be built into the mechanism. If the machine were to go down while messages were in the mailboxes, all data will be lost. In fact, if the system goes down, the IPC loses knowledge of the currently active mailboxes as well as the data. A machine is said to go down when only a full cold start will enable normal operation. If the mailbox is remote, then all communications through that mailbox are no longer possible. This will be indicated by a specific error code (see below).

1.4.1.3 Mailbox and User IDs for IPC

Mailbox IDs are unique to each system. This means that if two users have the same mailbox open, one user locally, one user remotely, then the mailbox IDs returned to each may not be the same. Never assume a mailbox id is anything other than the one returned by IPC$O (see below) for a specific mailbox.

A user's ID is unique to a system, then a mailbox. This means that the same user having two mailboxes open, one locally, one remotely, may have different user IDs for each mailbox opened. Anyone asking what this user's ID is will get different user IDs for the same user for different mailboxes. It would be safe to always get the user ID for required users on a per mailbox basis, and never assume a particular user has the same user ID for two different mailboxes.

A user ID is now unique while a system is up. There were previous prolems with this which are now corrected. These problems required drastic changes to the design and have imposed the previous mention ID restrictions. Please refer to the IPC Design Specification for the details of the changes.

1.4.2 Semantics/Meaning

The following sections discuss the commands and the callable program interfaces which help the user use the IPC mechanism.

1.4.2.1 IPC Commands

Three commands have been created to help the user create, delete, and close mailboxes.

1.4.2.1.1 MAKE_MBX

MAKE_MBX allows a user to create a mailbox. Specifically, it creates an access category with the rights specified by the user. A ".MBX" suffix is used when creating the mailbox name. This suffix will appear when the LD command is used. This makes it easier to determine when a mailbox has been defined.

The syntax for this command is

MAKE_MBX pathname access_control_list

pathname - standard file system pathname to which ".MBX" will be appended

access_control_list - standard ACL mechanism access control list

1.4.2.1.2 DELETE_MBX

DELETE_MBX allows a user to delete a mailbox. Specifically, it deletes an access category which had previously been set up to be a mailbox name, i.e., it deletes a mailbox name. This means that if a mailbox is in use at the time of the DELETE_MBX operation, only the name is deleted. The contents of the mailbox are not affected.

The syntax for this command is

DELETE_MBX pathname

pathname - standard file system pathname which points to an access category and ends in ".MBX"

1.4.2.1.3 CLOSE_MBX

CLOSE_MBX allows a user to either close a specific mailbox or close all of their mailboxes. Closing a mailbox gets rid of all pending messages for the user waiting to be read. Closing a mailbox does not get rid of the mailbox's associated access category, the mailbox name.

The syntax for this command is

CLOSE_MBX {mailbox_id}

mailbox_id - IPC mechanism's unique id for the mailbox to be closed

NOTE1: The mailbox_id argument is optional. If not

provided, all of a user's mailboxes will be closed.

NOTE2: The mailbox_id can only be obtained from the IPC mechanism via a program call. Therefore, to use this option, the user must obtain the mailbox_id from a program which calls IPC$O. Please refer to section 1.4.2.2 for further information.

1.4.2.2 IPC Program Interfaces

These modules will handle such areas as opening and closing a user's mailbox (similar to opening and closing a file), sending messages, receiving messages, controlling the ability to be software interrupted, and getting status about the IPC and its users.

The following interfaces have been defined to perform the above activities:

1) OPEN - Opens a mailbox after checking for the proper access rights and sets the mailbox either to be software interrupted or to use semaphore notification. Returns a mailbox id to be used as a shorthand means of telling the IPC which mailbox to use.

2) CLOSE - Closes a specific mailbox.

3) CLOSE_ALL - Closes all of a user's mailboxes.

4) SEND - Sends a message through the specified mailbox to a specific user.

5) SEND_ANY_USER - Sends a message through the specified mailbox to any user attached to the mailbox. Only one user gets the message.

6) SEND_ALL_USERS - Sends a message to all users attached to the specified mailbox.

7) RECEIVE - Receives a message from the specified mailbox. The message may have been addressed specifically by one of the send routines to the calling user or to any user.

8) RECEIVE_ANY_MAILBOX - Receives any message that may have been sent to this user. The message may come from any of the mailboxes to which the user is attached.

9) CONTROL - Turns on/off the notification that a message is waiting.

10) STATUS - Returns status about various parameters of the IPC mechanism or of a specific mailbox.

11) GET_USERS - Gets a list of all readers or writers or readers and writers attached to a specified mailbox.

The following sections fully describe these program interfaces.

1.4.2.2.1 IPC$O

Opens An IPC Mailbox

IPC$O		IPC$O

Subroutine TUE, 11 JAN 1983

Name: IPC$O

Purpose:

Opens a mailbox after checking for the proper access rights and sets the mailbox either to be software interrupted or to use semaphore notification. Returns a mailbox id to be used as a shorthand means of telling the IPC which mailbox to use.

Usage:

```
dcl ipc$o(fixed bin, fixed bin, char(128) var,
          fixed bin, fixed bin);

call ipc$o(access_key, notification_key, pathname,
           mailbox_id, ercode);
```

access_key - Access key. Examples are:

K$READ - Open mailbox for reading

K$WRIT - Open mailbox for writing

K$RDWR - Open mailbox for reading and writing

notification_key - Notification type key.
Ignored for K$WRIT.
Examples are:

K$NFSM - Use semaphores for notification

K$NFIN - Use software interrupts for notification

pathname - Pathname to access category which will be used as the mailbox name. It is also the object upon which the access rights to the mailbox are set and verified.

mailbox_id - Shorthand identifier to be used by the programmer when addressing this mailbox in all future IPC calls. (returned)

ercode - Standard PRIMOS error code. Examples are:

E$KEY - Bad access or notification key

E$NRIT - Insufficient rights to mailbox

E$FNTF - Mailbox (name) not found

E$FIUS - Mailbox already opened

E$ROOM - Not enough room to create mailbox

Other network errors.

1.4.2.2.2 IPC$C

Closes An IPC Mailbox

IPC$C		IPC$C

Subroutine TUE, 10 AUG 1982

Name: IPC$C

Purpose:

Closes a mailbox.

Usage:

```
dcl ipc$c(fixed bin, fixed bin);

call ipc$c(mailbox_id, ercode);
```

mailbox_id - Shorthand identifier which tells the IPC which mailbox to address.

ercode - Standard PRIMOS error code. Examples are:

E$NACC - Mailbox not accessable

Network errors will not be reported. Instead, the system will keep attempting to close the mailbox until the network comes back up or the remote system is cold started. Reopening the mailbox while it is in this state will not re-establish the mailbox remotely. It will merely flag the mailbox as being reopened.

1.4.2.2.3 IPC$CA

Closes All Of A User's Mailboxes

IPC$CA		IPC$CA

Subroutine TUE, 11 JAN 1983

Name: IPC$CA

Purpose:

Closes all the mailboxes that a user currently owns.

Usage:

```
dcl ipc$ca entry();

call ipc$ca();
```

1.4.2.2.4 IPC$SS

Send A Message To A Specific User

IPC$SS		IPC$SS

Subroutine TUE, 10 AUG 1982

Name: IPC$SS

Purpose:

Sends a message through the specified mailbox to a specific user.

Usage:

```
dcl ipc$ss(fixed bin, fixed bin, ptr, fixed bin,
           fixed bin);

call ipc$ss(mailbox_id, ipc_user_id, msg_ptr,
            msg_size, ercode);
```

mailbox_id - Shorthand identifier which tells the IPC which mailbox to address.

ipc_user_id - IPC user id of message's receiver. The IPC gives each user a unique id for each mailbox. This id may also be the caller's (see the IPC$GU subroutine).

msg_ptr - Pointer to the message to be sent.

msg_size - Size of the message in words.

ercode - Standard PRIMOS error code. Examples are:

E$NRIT - Wrong access rights (mailbox is read only)

E$NACC - Mailbox not accessable

E$BPAR - Message size exceeds maximum message size

E$ROOM - Not enough room in IPC to send message

E$UADR - Unknown addressee

E$RSIN - Remote system has initialized - the node the mailbox resides on has cold started.

Other network errors.

1.4.2.2.5 IPC$SA

Sends A Message To Any User

IPC$SA		IPC$SA

Subroutine TUE, 10 AUG 1982

Name: IPC$SA

Purpose:

Sends a message through the specified mailbox to any user attached to the mailbox. Only one user will get the message, the first receiver who reads it.

Usage:

```
dcl ipc$sa(fixed bin, ptr, fixed bin, fixed bin);

call ipc$sa(mailbox_id, msg_ptr, msg_size,
            ercode);
```

mailbox_id - Shorthand identifier which tells the IPC which mailbox to address.

msg_ptr - Pointer to the message to be sent.

msg_size - Actual size of the message in words.

ercode - Standard PRIMOS error code. Examples are:

E$NRIT - Wrong access rights (mailbox is read only)

E$NACC - Mailbox not accessable

E$BPAR - Message size exceeds maximum message size

E$ROOM - Not enough room in IPC to send message

E$UADR - No users attached to mailbox

E$RSIN - Remote system has initialized - the node the mailbox resides on has cold started.

Other network errors.

1.4.2.2.6 IPC$SSA

Sends A Message To Any User and Self

IPC$SSA		IPC$SSA

Subroutine TUE, 10 AUG 1982

Name: IPCSSA

Purpose:

Sends a message through the specified mailbox to any user attached to the mailbox. Only one user will get the message, the first receiver who reads it. This routine is essentially the same as IPC$SA except the calling user is also allowed to read the message if he has the mailbox open for reading.

Usage:

```
dcl ipc$ssa(fixed bin, ptr, fixed bin, fixed bin)
;

call ipc$ssa(mailbox_id, msg_ptr, msg_size,
             ercode);
```

mailbox_id - Shorthand identifier which tells the IPC which mailbox to address.

msg_ptr - Pointer to the message to be sent.

msg_size - Actual size of the message in words.

ercode - Standard PRIMOS error code. Examples are:

E$NRIT - Wrong access rights (mailbox is read only)

E$NACC - Mailbox not accessable

E$BPAR - Message size exceeds maximum message size

E$ROOM - Not enough room in IPC to send message

E$UADR - No users attached to mailbox

E$RSIN - Remote system has initialized - the node the mailbox resides on has cold started.

Other network errors.

1.4.2.2.7 IPC$SB

Send A Message To All Users (Broadcast)

IPC$SB		IPC$SB

Subroutine TUE, 11 JAN 1983

Name: IPC$SB

Purpose:

Sends a message to all users attached to the specified mailbox. This kind of message is also known as broadcast message.

Usage:

```
dcl ipc$sb(fixed bin, ptr, fixed bin, fixed bin);

call ipc$sb(mailbox_id, msg_ptr, msg_size,
            ercode);
```

mailbox_id - Shorthand identifier which tells the IPC which mailbox to address.

msg_ptr - Pointer to the message to be sent.

msg_size - Actual size of the message in words.

ercode - Standard PRIMOS error code. Examples are:

E$NRIT - Wrong access rights (mailbox is read only)

E$NACC - Mailbox not accessable

E$BPAR - Message size exceeds maximum message size

E$ROOM - Not enough room in IPC to send message

E$UADR - No users attached to mailbox

E$RSIN - Remote system has initialized - the node the mailbox resides on has cold started.

Other network errors.

1.4.2.2.8 IPC$SSB

Send A Message To All Users (Broadcast) and Self

IPC$SSB		IPC$SSB

Subroutine TUE, 11 JAN 1983

Name: IPC$SSB

Purpose:

Sends a message to all users attached to the specified mailbox. This kind of message is also known as broadcast message. If the caller has the mailbox open for reading, then the caller may also read the message.

Usage:

```
dcl ipc$ssb(fixed bin, ptr, fixed bin, fixed bin)
;

call ipc$ssb(mailbox_id, msg_ptr, msg_size,
             ercode);
```

mailbox_id - Shorthand identifier which tells the IPC which mailbox to address.

msg_ptr - Pointer to the message to be sent.

msg_size - Actual size of the message in words.

ercode - Standard PRIMOS error code. Examples are:

E$NRIT - Wrong access rights (mailbox is read only)

E$NACC - Mailbox not accessable

E$BPAR - Message size exceeds maximum message size

E$ROOM - Not enough room in IPC to send message

E$UADR - No users attached to mailbox

E$RSIN - Remote system has initialized - the node the mailbox resides on has cold started.

Other network errors.

1.4.2.2.9 IPC$R

Receive A Message From A Specific Mailbox

IPC$R		IPC$R

Subroutine TUE, 11 JAN 1983

Name: IPC$R

Purpose:

Receives a message from the specified mailbox. The message may have been addressed specifically to the calling user or to any user.

Usage:

```
dcl ipc$r(fixed bin, fixed bin, ptr, fixed bin,
          fixed bin, fixed bin, fixed bin);

call ipc$r(read_key, mailbox_id, buffer_ptr,
           buffer_size, msg_size, ipc_user_id,
           ercode);
```

read_key - Read key. Examples are:

K$RDWT - Read a message and wait if a message is not present

K$READ - Read a message but don't wait if a message is not present (E$NDAT will be returned if no data is present)

mailbox_id - Shorthand identifier which tells the IPC which mailbox to address.

buffer_ptr - Pointer to the buffer into which the IPC should write the message.

buffer_size - Size of the buffer in words.

msg_size - Number of words in message. (returned)

ipc_user_id - IPC user id of message's sender. The IPC gives each user a unique id. (returned)

ercode - Standard PRIMOS error code. Examples are:

E$BKEY - Bad read key

E$NRIT - No access rights

E$NACC - Mailbox not accessable

E$BFTS - Buffer too small to write message to (message still pending)

E$NDAT - No data found - another reader has gotten the message first or no messages are pending

E$RSIN - Remote system has initialized - the node the mailbox resides on has cold started.

Other network errors.

1.4.2.2.10 IPC$RA

Receive Message From Any Mailbox

IPC$RA		IPC$RA

Subroutine TUE, 11 JAN 1983

Name: IPC$RA

Purpose:

Receives any message that may have been sent to this user. The message may come from any of the mailboxes to which the user is attached.

Usage:

```
dcl ipc$ra(fixed bin, ptr, fixed bin, fixed bin,
           fixed bin, fixed bin, fixed bin);

call ipc$ra(read_key, buffer_ptr, buffer_size,
            mailbox_id, msg_size, ipc_user_id,
            ercode);
```

read_key - Read key. Examples are:

K$RDWT - Read a message and wait if a message is not present

K$READ - Read a message but don't wait if a message is not present (E$NDAT will be returned if no data is present)

buffer_ptr - Pointer to the buffer into which the IPC should write the message.

buffer_size - Size of the buffer in words.

mailbox_id - Identifier which tells the user from which mailbox the message has come.

msg_size - Number of words in message. (returned)

ipc_user_id - IPC user id of message's sender. The IPC gives each user a unique id. (returned)

ercode - Standard PRIMOS error code. Examples are:

E$BKEY - Bad read key

E$NACC - Mailbox not accessable

E$FNTF - Mailbox not found (user has no mailboxes open)

E$BFTS - Buffer too small to write message to (message is still pending)

E$NDAT - No message found

E$RSIN - Remote system has initialized - the node the mailbox resides on has cold started.

Other network errors.

Note: Any network errors that are encountered by the IPC will be considered the same as no messages pending and will not be reported. Only IPC$R is capable of determining if a network error is existant on a remote mailbox for receiving.

1.4.2.2.11 IPC$CN

The IPC mechanism will not have a specific module created to enable/disable the signal that is generated when a message is sent. This option will be provided for the IPC by the existing SW$INT routine. A selection bit (bit 7) has been defined within SW$INT for IPC. For further information on how to use SW$INT see Software Interrupt Control Module Functional Specification PE-TI-1005.

1.4.2.2.12 IPC$ST

Provides IPC Status

IPC$ST		IPC$ST

Subroutine TUE, 11 JAN 1983

Name: IPC$ST

Purpose:

Returns status about various parameters of a specific mailbox.

Usage:

```
dcl ipc$st(fixed bin, fixed bin, fixed bin,
           fixed bin);

call ipc$st(key, mailbox_id, value, ercode);
```

key - Action key. Examples are:

K$NMSG - Get number of messages in this mailbox for this user

K$MROM - Get maximum amount of space allowed in a mailbox (mailbox_id not checked)

K$ROOM - Get amount of space left in this mailbox

K$NUSR - Get number of users attached to this mailbox

mailbox_id - Shorthand identifier which tells the IPC which mailbox to address.

value - Item queried for. (returned)

ercode - Standard PRIMOS error code. Examples are:

E$NACC - Mailbox not accessable

E$BKEY - Bad key

E$RSIN - Remote system has initialized - the node the mailbox resides on has cold started.

Other network errors.

Note: K$MROM is the maximum space allowed to be used

by this mailbox. It is not neccessarily the maximum message size. The maximum message size is 3072 for networked and local mailboxes. This is a NPX restriction and is enforced at Rev 20.0.

1.4.2.2.13 IPC$GU

Gets A List Of Users Attached To A Specified Mailbox

```
----------                                        ----------
| IPC$GU |                                        | IPC$GU |
----------                                        ----------
```

Subroutine TUE, 11 JAN 1983

Name: IPC$GU

Purpose:

Gets a list of all readers or writers or reader/writers attached to a specified mailbox. The calling user's id is never returned.

Usage:

```
dcl ipc$gu entry(fixed bin, fixed bin, ptr,
                 fixed bin, fixed bin,
                 fixed bin);

call ipc$gu(read_key, mailbox_id, buffer_ptr,
            buffer_size, returned_size, ercode);
```

read_key - Read key. Examples are:

K$READ - Get user IDs capable of reading from this mailbox

K$WRIT - Get user IDs capable of writing to this mailbox

K$RDWR - Get user IDs capable of reading and writing to this mailbox

K$MINE - Get my user ID if mailbox is open for reading

mailbox_id - Shorthand identifier which tells the IPC which mailbox to address.

buffer_ptr - Pointer to the buffer into which the IPC should write the list of users.

buffer_size - Size of the buffer in words.

returned_size - Size of the list written into the buffer. (returned)

ercode - Standard PRIMOS error code. Examples are:

E$BKEY - Bad read key

E$NACC - Mailbox not accessable

E$BFTS - Buffer too small

E$RSIN - Remote system has initialized - the node the mailbox resides on has cold started.

Other network errors.

1.4.3 Errors

The subject of errors may be broken into two parts, error detection and error handling.

1.4.3.1 Error Detection

This IPC mechanism may detect two different kinds of errors, memory requirement errors and recoverability errors.

1.4.3.1.1 Memory Requirement Errors

Memory requirement errors may occur whenever the IPC system's use of main memory exceeds its allotted space, i.e. the IPC will only be given a fixed amount of memory in which to store its needed control information and the user's messages. These errors are soft errors. The IPC will be able to recover from them. They may go away the next time a message is taken out of the system or the next time a dynamic system segment is freed. (The IPC will use dynamically allocated system segments for its databases. It is not valid for the IPC to automatically delete messages to recover space. It does not have any means of determining which messages may even be deleted. Therefore, the user should attempt to receive messages if any are anticipated to be pending.) The key point is that once space is freed for use by the IPC, the IPC can again function normally. The recommended recovery for E$ROOM on sends is to retry the operation.

1.4.3.1.2 Recoverability Errors

Recoverability errors occur when the system halts and requires a cold start. The IPC will not be able to reconstruct the databases contained within before the halt occurred. Therefore, any data that was in the system will be lost. Users should not be dependent upon messages always getting to their receivers.

Another type of error is network error which is also a recoverabilty error. Transient network failures may be "ridden through". This means the server may retry the operation and may be successful if the network has temporarily failed. If, however, the remote node has gone down, then this cannot be distinguished from a temporary network failure. Only when the node has come back up can the IPC report a problem. When this occurs, the error code E$RSIN will be returned. If

this error is encountered, then the mailbox should be re-established. This is achieved by closing and reopening the mailbox.

Because of possible temporary network failures, notifies for networked mailboxes cannot be gaurenteed. A network failure may occur between the time the message is sent and the time the notify is delivered. This implies that a user may not receive a notify even though a message is pending, or conversely, a notify may occur but the message is inaccessable (due to network failure).

NOTE: Sending messages to a remote mailbox rapidly may cause the remote node to run out of slaves because the IPC allocates slaves on a per-call basis and does not maintain a slave for the duration of the mailbox's existance. This allows the programmer to recover from transient network failures. You may defeat this (or prevent this error) by establishing a slave on the remote system by using the file system (attaching or opening a file on the remote system) or using NPX directly. This also enhances remote mailbox performance.

1.4.3.2 Error Handling

Error handling must also be made an easy operation for the programmer. The system has a choice as to what to do when an error is detected:

a) It could do nothing and abort. This is not very reasonable when dealing with users.

b) It could hide the errors from the users.

c) It could try to correct the errors.

d) It could simply tell the users of the errors.

Which of these actions should be taken is a matter of debate. In any case, whichever solution is taken should allow the user to understand what is going on from within a program. The error actions should not assume that a user will be sophisticated enough to correct any mistake that might occur. The IPC should gracefully try to handle the situation as well as it can. If it must inform the user of an error, the error messages should be very clear. Enough for the user to take logical actions based upon the error messages (codes).

1.4.4 Restrictions

The proposed IPC mechanism is a prototype. It may change at any particular revision of PRIMOS. Therefore, any user of this IPC should be aware that they may have to recode their application whenever these changes occur.

The IPC does not guarantee any recoverability. Users should be aware that once a message is sent, there is no concrete statement made that the message will get to the receiver. No guarantees are made about the order in which the messages wiil be received. Guaranteed reception is the responsibility of the programmer and ordering may need to be checked.

The IPC will only be given a limited amount of memory space. Therefore, any users of the mechanism should realize that too many messages/mailboxes may cause the mechanism to run out of space. Along these lines, users should realize that other subsystems/application programs/system users may also be utilizing the functionality of the IPC. It may even be easier to run out of space than an isolated application may believe.

1.5 Testability

The IPC will be put together in an intelligent manner using the following in its design to aid debugging:

1) doubly linked lists

2) isolated network code

3) fully modular functionality

4) as much existing functionality as possible

The IPC will be tested thoroughly by creating a package of programs which will run as phantoms. Where networks are concerned, phantoms will also be used. It will be their task to exercise all of the various functions within the mechanism, therefore providing a full functional test.

2 DESIGN NOTES

It has been observed that the current version of PRIMOS (Revision 19) contains four features that can be extended or simply used by the proposed IPC: the Phantom Logout Notification Code, the semaphore mechanism, the condition mechanism, and NPX. The Phantom Logout Notification code would be used as an example of a subset IPC, the semaphore mechanism and the condition mechanism would be used for process synchronization, and NPX would be used to create the networking capability.

2.1 A Subset IPC

Presently a subset of an IPC has been implemented for Phantom Logout Notification. This subset is a kind of static mailbox system applicable to PRIMOS use. It does not contain any features that would allow for the notion of dynamically created mailboxes which reader's and writer's open and close nor does it allow the "all" and "any" functionality described above. It also does not provide any networking capability.

2.2 Process Synchronization

As far as the idea of synchronization is to be developed, semaphore operations, polling, and notification techniques will be implemented. Waits and notifies (synchronous synchronization) will be accomplished using the Prime 50 Series/PRIMOS semaphore operations. Signalling (asynchronous synchronization) will be accomplished using the PRIMOS implemented PL/1 signalling technique. Polling (synchronous synchronization) will be accomplished by the user calling the receive routines in some form of loop while notification has been turned off utilizing the K$READ key in the receive routines.

2.3 Networking Capability

NPX will allow the IPC to be designed to work across the network. The internal databases of the IPC will contain information about a mailbox. One of these pieces will indicate that a mailbox or user is remote.

In addition to the facilities currently in PRIMOS, the IPC will use a dynamic storage allocation mechanism which is currently built into PRIMOS. The Dynamic Storage Allocator Functional Specification PE-TI-1062 documents this mechanism.

3 RELIABILITY

The following are those items for which reliability can be stated at present:

1) The IPC should be as reliable as possible within the confines of a local node and within cold start to halt time.

2) It should not be guilty of allowing either deadlock or critical embrace to occur when access is made to its databases.

3) The IPC should not have to worry about creating faults when traversing its mailboxes.

4) All messages that are accepted into the system shall be protected from users who do not have correct access rights.

5) The IPC will not retry network failures, however, it will allow temporary network failures to be "ridden" through. For example, if a network error is returned from a read operation, then the read may be tried again at a later time, (ie, when the networks come back up).

4 PERFORMANCE

We will attempt to:

1) Speed up data transfer.

2) Speed up the open operation which has the highest cost. This is justified as in general not many opens will be done per mailbox.

3) Minimize the cost in both CPU time and paging from reads and writes. The operations of read and write should not be expensive as the data is to be stored in a shared system segment. With enough use of the IPC, its pages are likely to be found in main memory, and paging will not result.

We have no basis upon which to set limits for the operational speed of the IPC as it is a prototype. The hope is to be able to use it as a base for future IPC performances. Therefore attempts will be made to identify the IPC's bottlenecks and to measure how long it takes to perform an IPC data transfer.

5 CONFIGURATION

The IPC will come as a product within PRIMOS. It will not have any parameters that need to be configured at first release. Its memory requirements will be set at compile time. No operator actions will need to be performed.

Perhaps at a future release we may add certain configuration parameters that will effect such items as number of mailboxes, total space used by the IPC, etc.

6 INSTALLATION

No extra installation procedures are required since this IPC comes as part of PRIMOS.

7 EASE OF USE

Perhaps the most important requirement not yet mentioned about this IPC is that it must be easy to use. It should be easy to use for both application programmers and systems programmers. The program interfaces should be as simple as possible. Each type of call should not involve many different arguments, and the arguments should be simple in nature, such as the name of the pathway, what to write, where to write, and error status. There should not be many types of interface routines to call. The set should consist of operations that read information, write information, provide status information, and provide the means for opening and closing a pathway. These interfaces have already been defined in the section on Operational Procedures.

7.1 Example Of The IPC In Use

The following is an example of how server processes and their users would communicate via this IPC.

The basic premises are:

1) Servers control access to a secured data base.

2) Users need to get information to and from this data base.

3) Users act through a transaction program which is the agent that sends messages to the server.

During the design of the service the programmers must agree upon the name of the mailboxes (access catagories) needed and who is to have what kind of access to the mailboxes (access catagories). Before anyone is allowed to use the service these mailboxes (access catagories) must be set up.

The following two sections demonstrate how to use the IPC using both the semaphore or software interrupt synchronization techniques.

7.1.1 Using Semaphore Synchronization

When the servers are initially executed, they must perform two actions with respect to the IPC; open a mailbox from which they will read user's messages and open a mailbox for writing through which they will send responses back to the users. The following code performs these actions:

```
call ipc$o(k$read, k$nfsm, orders_mailbox_name,
           orders_mailbox_id, ercode);
```

```
call ipc$o(k$writ, k$nfsm, response_mailbox_name,
           response_mailbox_id, ercode);
```

After opening these mailboxes, the servers can then wait for an order. After an order is received, the servers must act upon it and send a response back to the requestor. Usually, one of the orders will be a shut down order. The following code performs these actions:

```
do while(<not shut down order>);
  call ipc$r(k$rdwt, orders_mailbox_id, buffer,
             buffer_size, msg_size, ipc_user_id,
             ercode);

  <process the order found in buffer>

  call ipc$ss(response_mailbox_id, ipc_user_id, msg_ptr,
              msg_size, ercode);
end;
```

When the transaction program is initially executed it also must perform two actions with respect to the IPC; open a mailbox for writing through which it will send orders to any server and open a mailbox from which it will read the servers' responses. The following code performs these actions:

```
call ipc$o(k$writ, k$nfsm, orders_mailbox_name,
           orders_mailbox_id, ercode);
call ipc$o(k$read, k$nfsm, response_mailbox_name,
           response_mailbox_id, ercode);
```

When a user tells the transaction program to do something, the transaction program must translate the request into an order and send the order to any server. The following code demonstrates this:

```
<translate the request>

call ipc$sa(orders_mailbox_id, order, order_size,
            ercode);
```

After sending the order the transaction program must wait for a response and then inform the user of the server's response. The following code demonstrates this:

```
call ipc$r(k$rdwt, response_mailbox_id, buffer,
           buffer_size,msg_size, ipc_user_id, ercode);

<inform user of server's response>
```

At this point the transaction program can ask for another order.

Note that the order is sent to any reader of the orders mailbox. This allows for more than one server to be attached to the orders mailbox.

7.1.2 Using Software Interrupt Synchronization

When the servers are initially executed they must perform three actions with respect to the IPC; set up an on-unit which will see any messages which are sent to a server, open a mailbox from which they will read user's messages, and open a mailbox for writing through which they will send responses back to the users. The following code performs these actions:

```
call mkonu$('IPC_MSG_WAITING$', orders_onunit);
call ipc$o(k$writ, k$nfin, orders_mailbox_name,
           orders_mailbox_id, ercode);
call ipc$o(k$read, k$nfin, response_mailbox_name,
           response_mailbox_id, ercode);
```

After performing the above operations, the servers must wait for orders to be sent or until a shut down order is received by the "order_onunit". While waiting the servers are free to perform any interruptable actions. The following code demonstrates this:

```
do while(<shut down order not seen>);
  call sleep$(1000000); /* do interruptable work here */
end;
```

Note: Instead of "sleeping" the server could perform any interruptable activity.

The "orders_onunit" is responsible for reading the user's messages, acting on the orders in the messages, and sending back the server's response. The on-unit is also responsible for turning notification off before receiving the message and on after it finishes processing the order. This should be done to prevent multiple notifications from interfering with each other. The form of the on-unit is as follows:

```
orders_onunit: proc(cfp);

  dcl cfp ptr;          /* condition frame pointer */

  call sw$int(...);     /* turn off ipc interrupt */

  do while(<more orders to receive>);
    call ipc$r(k$read, orders_mailbox_id, buffer,
               buffer_size, msg_size, ipc_user_id,
               ercode);


    <process the order found in buffer - if a shut down
     order communicate this to the main server program>


    call ipc$ss(response_mailbox_id, ipc_user_id,
                msg_ptr, msg_size, ercode);
  end;

  call sw$int(...);     /* turn on ipc interrupt */

end;                    /* orders_onunit */
```

Note that the on-unit will handle as many messages as are in the IPC for it once invoked. Messages may be placed into the IPC while the on-unit is executing. Remember, the on-unit may lose control getting a lock or by being time slice ended.

Also note that the on-unit may get invoked again because of a pending IPC interrupt that was set while the on-unit was executing. The on-unit may then find no data in the IPC. (It just handled the data while it was last running.)

When the transaction program is initially executed it also must perform three actions with respect to the IPC; open a mailbox for writing through which it will send orders to any server, open a mailbox from which it will read the server's responses, and set-up an on-unit which will see the responses and present them to the user. The following code performs these actions:

```
call mkonu$('IPC_MSG_WAITING$', response_onunit);
call ipc$o(k$writ, k$nfin, orders_mailbox_name,
           orders_mailbox_id, ercode);
call ipc$o(k$read, k$nfin, response_mailbox_name,
           response_mailbox_id, ercode);
```

When a user tells the transaction program to do something, the transaction program must translate the request into an order and send the order to any server. The following code demonstrates this:

```
<translate the request>

call ipc$sa(orders_mailbox_id, order, order_size,
            ercode);
```

After performing the above operations, the transaction program must wait for the server to send back its response. The following code demonstrates this:

```
do until(<response seen>);
  call sleep$(1000000);
end;
```

Note: Instead of "sleeping" the transaction program could perform any interruptable activity.

Note that the order is sent to any reader of the orders mailbox. This allows for more than one server to be attached to the orders mailbox.

The response_onunit is responsible for getting the response message from the server, setting it up for the user, and presenting it to the user: Its format is as follows:

```
response_onunit: proc(cfp);

  dcl cfp ptr;          /* condition frame pointer */

call ipc$r(k$read, response_mailbox_id, buffer,
           buffer_size, msg_size, ipc_user_id, ercode);

<process the response found in buffer and tell
transaction program that a response has been seen>

<present the response to the user>

end;                    /* response_onunit */
```

Note that no enabling/disabling of the IPC notification is needed. In this example, only one response will be seen by one user at a time as only one order is sent at a time.

7.2 Analysis Of Semaphore Vs. Interrupt Notification

An analysis of this example shows that it is much easier to code using the semaphore synchronization technique rather than the interrupt technique. All the code appears in the normal path of the programs rather than in an on-unit. Additionally, when using on-units, the program and the on-unit must communicate via global variables such as <response seen>.

8 MAINTAINABILITY

This IPC will be built as a completely self-contained mechanism to be added to PRIMOS. All maintenance for the product will be performed by the PRIMOS group. The code will be well structured and well documented. A Design Specification will be written to aid future PRIMOS group members whenever they are studying, debugging, enhancing, or just simply curious about the IPC. This specification may also be used by future developers of a fully complete IPC as a reference. A stand alone data base tracing program will be written to aid in debugging. Its functionality will be added to AUTOPSY.

9 COMPATIBILITY

As this IPC is a brand new product, one that has never before been introduced to PRIMOS, compatibility is not an issue.

It should be mentioned that future IPC mechanisms may wish to utilize some of the same interface definitions and/or control data structures.

Finally, it must be stated that future IPCs guarantee NO compatability with this prototype nor does this prototype guarantee any compatability with any future IPCs.

10 STANDARDS

The only standard that must be addressed by this IPC is the entry name standard. Each entrypoint IPC module has no more than six characters in its name. Therefore, they may be called via PRIMOS' dynamic linking mechanism. Each IPC interface module will begin with "IPC".

All internal OSSD coding standards will be followed except as pointed out in the IPC Design Sepcification PE-TI-XXXX.

11 REFERENCES

No specific outside literature was referenced when defining the interfaces to be included within the IPC. The literature mentioned in sections 1.2 and 1.4.2.2.9 was used when defining functionality.